# Specification and Analysis of Service Oriented Architectures within the Calculus of Communicating Sequential Processes (CSP)

Al-Homaimeedi, Abiar Suliman

*Awarding institution:*
King's College London


# Specification and Analysis of Service Oriented Architectures within the Calculus of Communicating Sequential Processes (CSP)

by

Abiar S. Al-Homaimeedi

A thesis submitted in partial fulfilment for the
degree of Doctor of Philosophy

Department of Informatics
Kings College London, University of London

Jan, 2015

# Acknowledgement

In the Name of Allah, Most Gracious, Most Merciful, all praise and thanks are purely due to Him.

It is my pleasure to express my special appreciation and thanks to my supervisor, Prof. Maribel Fernandez, for the continuous support of my Ph.D study and related research, for her patience, motivation, and immense knowledge. Her invaluable guidance helped me in all the time of research and writing of this thesis.

I would, also, like to express my special thanks to my previous supervisor, Dr. Iman Poernomo, who gave me the first chance to start my Ph.D and for his support during my first year.

Besides my supervisors, I would like to thank my thesis examiners: Prof. Cosimo Laneve and Prof. Alessio R. Lomuscio, for their insightful comments and encouragement, and for letting my defense be an enjoyable moment, thanks to them.

I would like to express my appreciation to my group members and colleagues in the Department of Informatics at Kings College London and in the Department of Information Technology at King Saud University. All of you have been there to support me during the preparation of this thesis. A special thanks goes to Peter Young for proof reading chapters of my thesis.

Words cannot express how grateful I am to my son Khalid and my husband Waleed for all of the sacrifices that they have made on my behalf. For all their patience and support.

A special thanks to my family. A special gratitude I give to my parents, your prayer for me was what sustained me thus far. I would also like to express my deepest appreciation to my brothers and my sisters who supported, and incented me to strive towards my goal.

Furthermore, I would like to thank all of my friends specially Eatedal and Lamya for their support and encouragement.

At the end, I would like to acknowledge with appreciation the crucial role of Mrs. Mary Dunning for proof reading this thesis.

# Abstract

Software architecture evolved from the monolithic paradigm to the Service-Oriented Computing (SOC) paradigm. IT systems in the SOC paradigm are based on service compositions. A service composition is an aggregate of loosely coupled autonomous heterogeneous services which are collectively composed to implement a particular task. Internet standards are the dominant modelling methods of SOC systems. However, they raise fundamental issues: standards lack formalism, and they fall short when being applied independently. The former issue has been solved and rigorous semantics have been developed for the different standards. However, the latter issue has only partially been solved, by developing new formal modelling languages that are adopting the concepts rather than the notations of the internet standards. In principle, the main concepts that should be hosted in SOC modelling languages are: asynchronicity, mobility, multiparty sessions, and compensations. However, not all of these concepts are supported in the current developed modelling languages. This thesis addresses this problem and proposes a new formal modelling language for SOC systems which is adequately expressive to model the previous concepts. Additionally, the thesis provides an implementation for the new modelling language in a model checker to facilitate automated formal reasoning on systems properties like: good/bad traces, deadlock-freedom, and livelock-freedom.

**keywords:** Service-Oriented Computing, modelling languages, process calculi, CSP, sessions, mobility, asynchrony, transitionality, long running transactions, compensations, channels, buffers, invocations, services, service composition

# Contents

# List of Figures

# Notations

Throughout this thesis, the following notations are used:

| | |
|---|---|
| $p, q, \ldots$ | denote processes |
| $pp, qq, \ldots$ | denote compensable processes |
| $\Sigma$ | is the universal set that contains all the observable events in a system |
| $a, b$ | are used to range over $\Sigma$ |
| $\Omega$ | is the set of terminal events |
| $\omega, \omega'$ | are used to range over $\Omega$ |
| $\Sigma^{\tau\Omega}$ | is the universal set $\Sigma^{\tau}$ union $\Omega$ |
| $\Sigma^{\tau}$ | the universal set $\Sigma$ and in addition the silent event $\tau$ |
| $\Sigma^{\surd}$ | the universal set $\Sigma$ and in addition the termination event $\surd$ |
| $\Sigma^{\tau\surd}$ | the set $\Sigma \cup \{\tau, \surd\}$ |
| $A, B$ | sets of observable events |
| $R$ | denotes renaming relation |
| $s$ | denotes lists |
| $R$ | renaming relation |
| $\langle x \rangle$ | denotes a list which has the element $x$ |
| $\alpha p$ | denotes the set of events that process $p$ can perform. |

# 1 Introduction

The Service-Oriented Computing (SOC) paradigm refers to the set of concepts, theories and techniques that represent computing in Service-Oriented Architecture (SOA), in which software applications are constructed based on independent program modules, namely services.

In essence, every service should perform a single task. This task could be a simple one like checking if a number is prime, or a complex task like an airline reservation system. The internal logic of these services is hidden from their environment. Instead, services provide a standard interface, which expresses the service functionality and defines the way of communicating with the service.

SOC applications, namely service compositions, are *sets of services* which collaborate according to a *predefined scenario* in order to accomplish a single task. As the name implies, collaboration between services is triggered via service request messages. In a service composition, collaborating services generally communicate only by passing messages, and these services could be heterogeneous, i.e. provided by different organizations or running on different platforms [67].

The initial set of services in a composition are selected according to the function that they offer, and how it will contribute to accomplishing the task of the composition. However, maintaining a stable fixed service composition is not realistic in dynamic business environments. In such environments, business requirements could frequently change [130]. Thus, during the life time of a composition, the initial set of services can be dynamically changed due to several technical or economic reasons. For instance, the old service being no longer available or the new service being cheaper.

Basically, in the SOC paradigm, services in a composition could be allocated and selected as follows:

- First, the service should be available to be selected. For that, the service provider implements the service logic and interface. Following that, the provider publishes the service online by placing the service interface in service registries, i.e. a private or public directory in a server within a network where the interface of new services could be stored, searched, and retrieved [67].
- Secondly, the service requester in a composition, usually called the client partner, searches service registries to seek out services whose interfaces meet the client's requirements. If allocated, a subscribe request is sent to the service providers. If approved, an agreement contract, namely Service Level Agreement (SLA), is conducted between the service requester and the service provider.

From the above, a Service Oriented Architecture can be represented as an interaction between three SOC entities: service requester, service provider and service registry as shown in Figure 1.1.

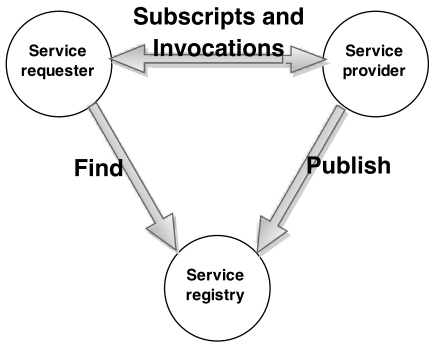


Figure 1.1: Service Oriented Architecture

If a set of services is initially selected in a service composition then the scenario for this composition can be defined. In the SOC paradigm, the scenario of a services composition, namely work flow, could be defined by two methods:

1. **Orchestration**: Orchestration scenarios in service composition are written from the perspective of each service. That is, each service expresses how it will

orchestrate with the other services in the composition. This includes demonstrating the sequence of messages to be exchanged, and the name of services to be invoked, i.e. asked for a service. Original, orchestration scenarios should have a leader service, namely an orchestrator, which controls the sequences of messages to be exchanged in a service composition. However, as this is not always the case, the work flow in a service composition is considered to be orchestration if the services lead other services by triggering their execution by means of invocations [30].

2. **Choreography**: Choreography scenarios demonstrate the global behaviour of a service composition as a whole, by expressing the interaction sequences that might take place in the composition. To have a working version of this scenario, usually the global view of a composition is projected to smaller endpoint views. The endpoint view captures the interactions of a single service.

Working with SOC offers significant advantages such as reusability as services can be used many times in different models. In addition, SOC models can be reconfigured dynamically as new services can join/leave the composition at run time. Integration with legacy systems is now easier due to the interaction achieved by exchanging messages and through standard interfaces. This will result in time and cost reduction in the process of software design [67, 130].

The SOC paradigm has different working instantiations like CORBA [2], DCOM [3], and J2EE [4]. However, one of the successful instantiation of SOC is the paradigm of web services. Web services abstract the interactions from the notion of objects, and centre interactions around services' request/ response messages.

Web services are supported by major computer corporations, including BEA, IBM, Microsoft, and Oracle. A set of internet standards has been developed to support and to regulate web services applications. The set includes: BPEL [14], the web orchestration language; WSDL [15], the web language for defining interfaces for services; SOAP [9], the web standard for defining and exchanging messages; and WSCDL [13], the web language for defining choreography scenarios.

Internet standards are XML-based languages which adapt the simple tag style for writing specifications. However, they raise the following issues:

1. Internet standards lack formal semantics. Therefore, analysing the correctness of a specification written using internet standards is not supported. In SOC, using the old testing methods is not realistic due to the dynamic nature of such systems. Moreover, testing does not guarantee the absence of errors.

   Lacking formal semantics also allows different interpretations of syntax primitives. For instance, the available execution engines, i.e. interpreter, for BPEL specifications, have different implementations for BPEL syntax. It has been shown in [132] that the three BPEL engines: ActiveBPEL [10], Apache ODE [1], and Oracle BPEL Process Manager [6], realize (execute) BPEL specification in different ways.

   The proposed solution for tackling this issue was to map specifications written onto these standards to a formal method for validation purposes. As shown in Figure 1.2, the design phase of service compositions is extended to incorporate verifications. Therefore, the validity of models can be formally verified by reasoning on the model properties desired. This solution has been adequately studied in the literature and several models have been proposed like [112, 83, 144, 90], which map internet standards, BPEL in particular, to different formal models.

2. Considering the expressiveness of these standards, they do not entirely capture the essence of the SOC paradigm. For instance, while a service composition has more than two services, sessions in a service composition can be conducted between two parties only. As a result, extra messages are required to circulate data between services in a service composition. Moreover, some designing aspects of these standards can be enhanced. For instance, the BPEL language exception system overcomes errors by using predefined handlers. To promote the dynamic nature of such models these handlers could be constructed dynamically at run time.

   To overcome the second concern without ignoring the first issue, a new formal modelling language should take the place of these standards in specifying SOC models. The new formal language should host the concepts rather than the constructs of the standards, then extend the concepts to possess better capabilities. For instance, hosting the invocation concept and extending the concept

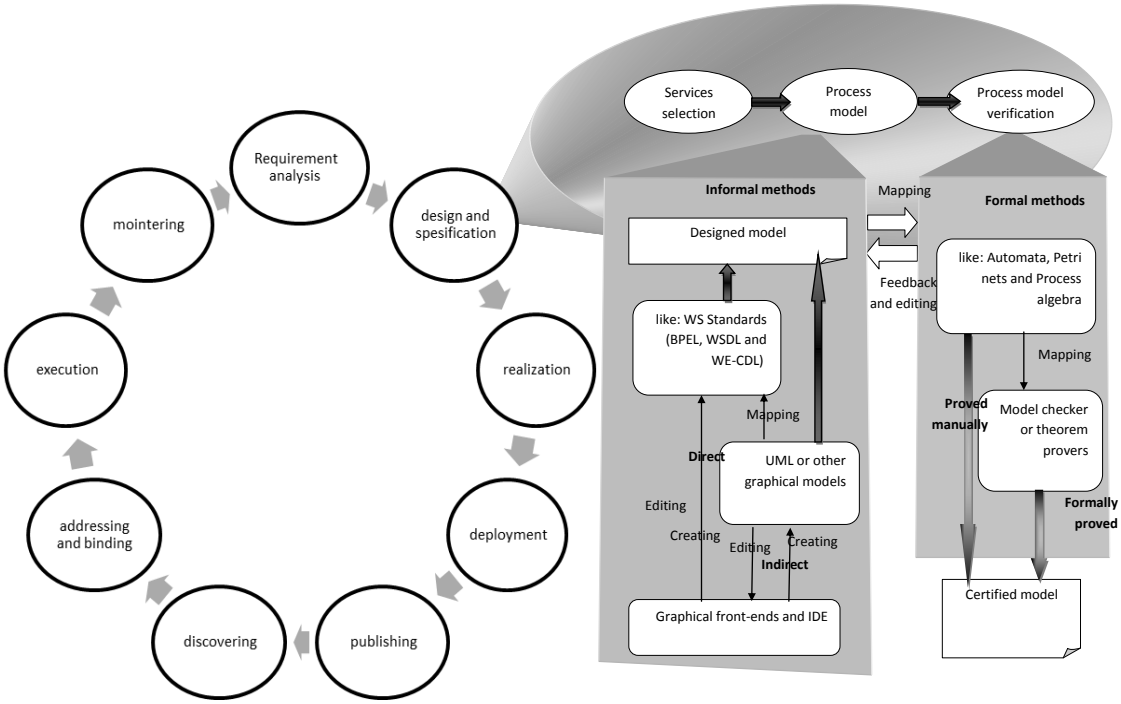


Figure 1.2: Life cycle of software processes in SOC paradigm.

to allow multiparty sessions.

In the literature (see Chapter 2 for details), a number of proposals have suggested formal modelling languages to specify SOC models. However, we consider this issue has yet to be fully solved, and we identify several areas for improvement as will be explained below. Therefore, in this thesis, we propose soaCSP process calculus as a new formal modelling language which can be used to design SOC models directly with enhanced designing primitives (see Chapter 6.2.2 for details), or to verify BPEL scripts (see Chapter 5 for details).

We identify below areas for improvement. In Chapter 2, we survey and discuss the previous works in process calculi for modelling service compositions in respect of these areas for improvement:

1. Formal modelling language which supports mixed synchronous and asynchronous

communications. This will provide a model where synchronous and asynchronous communications can be designed directly, and will simplify reasoning on these mixed communications.

2. Formal modelling language which supports mobile communications where connections can be exchanged between services within one session or different sessions. This will provide a model where sessions can be merged and communicate with processes outside the session boundaries.

3. Formal modelling language which supports multiparty sessions, where data can easily be circulated between services, and scenarios like broadcasting and multicasting is permitted. Sessions should be initiated by invocations only, so no need for further notions.

4. Formal modelling language which is supported by a computer-based framework to automate reasoning on properties of models.

5. Formal modelling language which is supported by an exception system which provides dynamically constructed handlers for overcoming system failures.

The new language proposed in this thesis, soaCSP, is considered to be an orchestration language as it is endowed with services invoking and publishing primitives. It should be adequately expressive to create, maintain, and terminate mutliparty sessions.

From the design perspective, orchestration work flows are implementable whereas choreography work flows need some sort of preparation before they have implementable scripts. For instance, projecting choreography work flows onto orchestration scripts has been done in [53]. Another point to consider is the scalability of choreography models. As choreography captures the global view of the system, the system should be complete. On the other hand, orchestrations' work flow is easily extendible, as it captures the view of individual services.

Our modelling language is founded on the process calculus of Communicating Sequential Processes (CSP). In essence, process calculi [125] are formal languages with relatively small numbers of constructs and operations which form the syntax of the calculus. The syntax of a process calculus is supported by formal semantics

to explain the meaning of its constructs and operations. This semantics can be: operational, defining the behavioural semantics of a calculus; denotational, defining the effects of these constructs and operations; and axiomatic, which defines the proof methods for asserting terms in a calculus [23]. The semantics of these languages are treated in a mathematical way to give a singular meaning to every sentence in order to facilitate the computational reasoning expected from such calculi.

We choose process calculi from among the other formal methods like Petri Nets [82] and automata [82] because these calculi are built around the principle of compositionality, in which the calculi semantics is given structurally so that the behaviour of the whole system is a function of the behaviour of its subsystems. Compositionality is well suited to orchestration languages, as service compositions' behaviours are described as the orchestration of the individual services' behaviours. Additionally, reasoning with process calculi can verify a range of desirable properties including: safety (assertion that an undesirable event will never happen), liveness (assertion that a desirable event will happen eventually) [23], and behavioural equalities, which could be useful if one process will replace another.

From the process calculi family we choose CSP for the following reasons:

- The design of the CSP *parallel composition*: in CSP, all processes participating in a parallel composition synchronise on a predefined set of events (*the interface set*). For instance, let $p, q, r$ be CSP processes, then in the parallel composition $(p \parallel_{\{\mid a,b \mid\}} q \parallel_{\{\mid a,b \mid\}} r)$, the processes $p$, $q$, and $r$ synchronise on events $a, b$ only, the rest of the events are evaluated independently. This design supports two features that we extend further in the thesis in order to achieve the desirable final results:
  - The parallel composition is parameterised with an explicit set which governs the synchronisation between participants. In the thesis we extend the CSP operational semantics to control the contents of this set in order to achieve mixed and mobile communications.
  - All participants in the parallel composition synchronise on shared events which are included in the interface set. In the thesis we extend the CSP operational semantics in order to facilitate communications in multiparty sessions.

- CSP is supported by a model checker namely Failure-Divergence Refinement (FDR): FDR implements the mathematical machinery and the theory of refinement that Hoare built for reasoning on the external behaviour of systems; see Chapter 2. It provides simple proof techniques for asserting the conformance between specifications and implementations, deadlock-freedom, and divergence-freedom, in addition to determinism and bisimulations (we illustrate some of these features in the case study in Section 7.3).

We support our language with an operational semantics which explains the behaviour of our language model. Although the foundation semantics of CSP are denotational semantics, in this thesis we use the operational semantics for the following reasons:

- We believe that listing computation steps, conditions, and conclusions shows clearly what the constructs do in a simple way which facilitates the understanding and analysis of the calculus.
- The computation steps facilitate the implementation of our language in the input language of FDR.
- Denotational semantics for a calculus can be derived from its operational semantics as is explained in [121].

In the following section we list the main contributions of this thesis. A detailed contributions list (if available) will be provided at the beginning of each chapter.

## 1.1 Thesis Contributions

The main contributions of this thesis are:

1. Develop a new formal modelling language for SOC systems based on the CSP process calculus, namely soaCSP. The new language is designed to model functional characteristics of SOC systems, in particular, it supports:

- *Mixed communications systems*, where messages can be sent synchronously, asynchronously and in interleaving mode. This is achieved by introducing asynchronous communications into the mixed synchronous/interleaving communications model of CSP in Chapter 3.
- *Dynamically adaptive systems*, where the linkage networks of services in a service composition are subject to change dynamically during run time. This is achieved by introducing mobility into CSP in Chapter 4.
- *Session-based communications systems*, where multi-party sessions can be created between services and clients with the following features:
  (a) Invocations to services create new sessions which are initially two-party sessions. Following that, further invocations in the sessions can be *new invocations* which create sub-sessions, or *joint invocations* which add the invoked service to the current session.
  (b) Service inside a session can simultaneously circulate information to more than one party. This includes broadcasting communications and multicasting communications.
  (c) Services in a session can communicate with other services in the parent session (if exists), or with other system processes outside the session. System processes can be defined as well, in the language, because we keep the standard syntax of CSP.
  (d) Creating sessions and terminating sessions are achieved in the semantics without users intervention.
  (e) Services in a session can interrupt their execution if one of its siblings in a session terminates.
  (f) Sessions can be temporarily merged with other sessions or other processes in the system.

  Session-based communications with the above features are achieved by labelling CSP communications with session keys in Chapter 5, and labelling asynchronous and mobile communications in Chapter 6.

2. *Develop a framework to formally describe and analyse soaCSP within the model checker of CSP* (FDR[12]). This is achieved by introducing new functions into

the syntax of $\text{CSP}_M$ (the input language of FDR) in Chapter 7.

3. *Improve the compensating version of CSP with dynamic compensations.* That is, delay the decision on compensations until the runtime of the system. This achieved by introducing variable compensations into compensating CSP in Chapter 8.

## 1.2 The Thesis Structure and Outputs

The thesis is organised as follows:

**Chapter Two: Background** This chapter presents the background information which highlights the motivation and the importance of the thesis. The chapter starts with a motivation example. Following that, a general overview of SOC formal modelling languages and SOC verification tools is provided. Finally, the chapter explains the preliminary theories that have been used in the thesis which include: BPEL language, CSP process calculus, and the $\pi$-calculus.

**Chapter Three: Asynchronous CSP (CSPa)** This chapter formally introduces asynchronous primitives into CSP.

**Chapter Four: Mobile CSP ($\mathcal{M}$CSP)** This chapter formally introduces mobility into CSP.

**Chapter Five: Session-Based CSP (CSPs)** This chapter formally presents session-based CSP.

**Chapter Six: CSP for Service-Oriented Architecture (soaCSP)** This chapter presents the complete calculus which supports the features developed in Chapters 3, 4, and 5.

**Chapter Seven: Model Checker for soaCSP** This chapter extends the syntax of $\text{CSP}_M$ [128] (the input language of the model checker) with new functions which implement the features of soaCSP. The chapter also illustrates the usability of the

calculus by implementing the finance case study [65, 66, 132] from Sensoria project [8] into soaCSP, and reason on the correctness of the implementation.

**Chapter Eight: Compensating CSP (cCSP)** This chapter formally presents the improved version of compensating CSP.

**Chapter Nine: Conclusions and Future work** This chapter concludes the thesis and demonstrates the possible extensions and future directions of the thesis content.

The contents of this thesis have been presented in several scientific events as Section 1.2.1 shows, and published in several papers as Section 1.2.2 enumerates.

### 1.2.1 Presentations

The content of Chapter 2 was presented in the following:

1. *Survey on the Service-Oriented Architecture Process Calculi*, the 6th Saudi Student conference, 2012, London.

The content of chapter 3 was presented in the following:

1. *Enabling Synchronous and Asynchronous Communications in CSP for SOC*, 9th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014), Universidade de Brasilia, Brasilia D.F., Brazil.

2. *Modelling Asynchronous Communication within the Process Calculus of Communicating Sequential Processes (CSP)*, the 8th Saudi Student Conference (8th SSC), 2015, Imperial College, London, UK.

The content of chapter 4 was presented in the following:

1. *Introducing Mobility into CSP*, Mobility Reading Group, 2014, Imperial College, London.

2. *Introducing Mobility into CSP*, The 26th Nordic Workshop on Programming Theory (NWPT 2014), 2014, Halmstad University, Halmstad, Sweden.

3. *Achieve pi-calculus Style Mobility into CSP*, The 29th British Colloquium for Theoretical Computer Science (BCTCS 2013), 2013, University of Bath, Bath, UK.

The content of Chapter 5 was presented in the following:

1. *Enhancing the Specification and Verification Techniques of Multiparty Sessions in SOC*, The 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015), 2015, University of Siena, Siena, Italy.

The content of Chapter 6 was presented as a poster in the following events:

1. *Ada Lovelace Day, Celebrating Women in Science*, 2014, King's College, London.

2. *7th Saudi Student Conference*, 2013, The University of Edinburgh, Edinburgh, UK.

3. *Research Day of the School of Natural & Mathematical Science*, 2013, King's College, London, UK.

The content of Chapter 8 was presented in the following:

1. *General Dynamic Recovery for Compensating CSP*, the 8th International Workshop on Developments in Computational Models (DCM 2012), 2012, University of Cambridge, Cambridge, UK.

2. *Modelling Dynamic Compensations within the Process Calculus of Communicating Sequential Processes (CSP)*, the 7th Saudi Student Conference (7th SSC), 2013, The University of Edinburgh, Edinburgh, UK.

### 1.2.2 Publications

1. The content of Chapter 3 and part of Chapter 7 were published in [20].

2. The content of Chapter 4 and part of Chapter 7 were published in [19].

3. The content of Chapter 5 and part of Chapter 7 were published in [21].

4. The content of Chapter 8 was published in [18].

# 2
# Background

In this chapter we present the background information which highlights the motivation and the importance of the thesis. The chapter starts with an overview of previous work in this area, followed by an example to motivate our work: a bank system where a client asks for a service, and to accomplish this service the bank system will establish a session between the client and the bank system. The session is used to avoid interference between this client interactions and other clients interactions. To resolve some of the decisions in this system a bank employee should be consulted in a time suitable for him. If the system receives the employee decision, then it will notify the client with the final result.

In this example we show how soaCSP provides the modelling primitives which allow the designer of this bank system to simply design services or processes. Services are the computation models which should work under a session like the bank service, on the other hand, processes are the computation models which work independently of any session like the bank employee. soaCSP also provides the means for these services and processes to communicate synchronously or asynchronously, where two or more processes, two or more services, or a mix of processes and services can communicate synchronously or asynchronously as the example shows.

In addition to the motivation example, this chapter includes a final section recalling the definitions and theories which are necessary to understand the rest of this thesis.

**Structure of this chapter** Section 2.1 and Section 2.2 provide a general overview of the previous works in SOC formal modelling languages and SOC verification tools respectively. Section 2.3 motivates the work in the thesis by an example. Finally, Section 2.4 presents the basic definitions and theories which are used in the thesis.

## 2.1 Previous Works In Process Calculi for Modelling Service Compositions

Surveying all the process calculi which contributed to SOC is not feasible due to the huge number available. Therefore, in this section we concentrate on orchestration calculi and we refer to other calculi when appropriate.

Historically, process calculi have been proposed to reason on concurrent systems since the 1970s. Calculus of Communicating Systems (CCS) [107] was the first in 1973 then Communicating Sequential Processes (CSP) [84, 129, 121] in 1976 followed by Algebra of Communicating Processes (ACP) [24] in 1982.

These well-known process calculi are insufficient for SOC models, because they lack primitives like sessions. However, they or their variations have been used in some SOC projects as underlying formalisms rather than modelling languages (c.f. [90, 125, 144]). In addition, these calculi have been used as a basis for more sophisticated calculi addressing SOC requirements.

In the following, we survey the scientific literature to present the state-of-the-art in the area of SOC calculi. However, given the amount of research in this area we classify them according to our area of interests.

Five criteria have been proposed in [114] to evaluate and compare formal methods for SOC systems. These criteria are: how the service is described in terms of functional, non-functional and behavioural specifications, the level of automation, and the ability to reason on the correctness of the model. In this chapter, we classify the literature according to these criteria, but we eliminate the non-functional and behavioural specifications as they are outside the scope of this thesis. We also further classify the criterion of modelling functional requirements into the areas of improvements that we proposed in Chapter 1.

We start by presenting proposals which have contributed to multiparty sessions,

then mobility. Following that, we discuss mixed asynchronous/ synchronous communications in SOC calculi. Transitionality is discussed after that. Finally, in the next section, we present the available verification tools and discuss the type of properties that could be checked by them.

We note with appreciation the Sensoria [8] project as it activated the research in this area. Several proposals came out of this project. Sensoria is an Integrated Project funded by the EU which lasted for 48 months and ran from 2005 till 2010. The aim of the project was "to develop a novel comprehensive approach to the engineering of service-oriented software systems where foundational theories, techniques and methods are fully integrated in a pragmatic software engineering approach, supporting semi-automatic development and deployment of self-adaptable (composite) services" [142].

### 2.1.1 Multiparty Sessions

General purpose mobile calculi (like $\pi$-calculus) are used as a base for more sophisticated SOC calculi. One of the first SOC calculi is the Service Centered Calculus (SCC) [35]. The SCC extends the $\pi$-calculus with the notion of sessions, which is basically a new primitive to create a private channel for exchanging messages between two participants sharing a public channel. The notion of sessions was originally adopted from the concept of pipelines in Orc [91]. Orc (for orchestration) is a concurrent programming language, which was firstly developed as a process calculus. Orc provides an orchestration construct to process different sites concurrently with management of time-outs, priorities, and failures but without introducing mobility [91].

Stream-based Service Centered Calculus (SSCC) [93] is presented as an extension of SCC with streams. Streams in SSCC act as orchestrators for a group of sessions. Yet another evolution from SCC is the Calculus of Sessions and Pipelines (CaSPiS) [36]. CaSPiS enhances SCC sessions to include pipelines which facilitate communicating with a service outside the two-party session. Additionally, CaSPiS has enhanced the closing algorithm of SCC by introducing an explicit primitive to kill sessions from the child services. This will ensure graceful termination, i.e. services can terminate if their siblings terminate. CaSPiS is further extended into MarCaSPiS [62] which is equipped with primitives to quantify Quality of Service (QoS) non-functional re-

quirements.

The Service-Oriented Computing Kernel (SOCK) [80] is a SOC calculus which is strongly inspired by BPEL, WS-CDL and WSDL. SOCK calculus has been divided into three layers: service behaviours, service engines (where sessions of services behaviours are grouped according to state and correlation set), and service systems (which catch the compositions of service engines equipped with locations). SOCK follows the internet standards' notations strictly which in turn affects its simplicity [93]. SOCK has been further extended with primitives to facilitate general dynamic recovery [79] which will be explained in Section 2.1.4. Later, in [97], the authors encoded dynamic Sagas into SOCK to introduce a new compensation mechanism into Sagas. SOCK has also been extended with a JAVA interpreter engine, namely Java Orchestration Language Interpreter Engine (Jolie) [5], and this project is still running.

The previous calculi establish a two-party session where communication can take place between two services only. However, improving the notion of session to handle communications between more than two services, i.e. a multiparty session, is essential in SOC service compositions where more than two services might need to communicate.

Multiparty sessions have been discussed in several papers that aim to develop new models to enhance the specification and verification techniques of multiparty sessions.

One of these works is the Conversation Calculus (CC) [137] which is an evolution from SCC. CC introduces an abstraction form of sessions, namely *conversations*, where multiparty sessions can be created by delegating communication to other services. This feature has been encoded by passing "conversation endpoints" to the targeted service so this service can join the session. If a service joins a session, it means that this service can communicate with any service in the session. However, only two services can communicate at a time, i.e. multicast and broadcast communication is not permitted. Developers of CC tried to encode cCSP (see Section 2.1.4) into CC [51] to introduce dynamic compensations but the result was not compositional (refer to [96] for more information). The Conversation Calculus has also been extended with time in [103].

Multiparty sessions have also been discussed in $\mu se$ calculus [45], where the same concept of *conversation endpoints* is used to allow multiple services to join the same

session sharing the same "session endpoint". $\mu se$, as CC, does not admit multicast and broadcast communications. The novelty in $\mu se$ is the merging primitive which permits two running sessions to be permanently merged dynamically. Moreover, in $\mu se$, services can be installed in *sites*. Services sharing the same sites can communicate without establishing sessions. It has been explained that services installed in the same site can represent services running in the same location, and therefore they can communicate and exchange data locally without establishing sessions.

The concept of *locations* has also been discussed in [98], where two channels in services can communicate and trigger further actions if they are co-located. This work was motivated by services implementation. The strong influence of locations on communications is not a concern of this thesis.

One of the earliest calculi developed within Sensoria was the Calculus for Orchestration of Web Services (COWS) [100]. The COWS constructs and operators were inspired mainly from BPEL, therefore, it only supports two-party sessions.

The Service-Oriented calculi presented until now were extensions of the $\pi$-calculus with different primitives to capture the essence of SOC interactions. These primitives can be seen as enforcement of communication patterns over the free communication offered by the base calculus ($\pi$-calculus). Another line of research introduces session types [56] to track the types of messages exchanged in each communication session as a plan for conversation.

In [52, 106, 75], session types are used to govern communications in two-party sessions. Later in [34], the concept of multiparty sessions has been introduced with the help of session types. However, multiparty sessions, in this research, are created between one master endpoint and one or more slave endpoints, and direct communications are only allowed between the master and any slave endpoints.

Inspired by the work in [34], a calculus of session types has been studied in [87]. The calculus is based on the choreography metaphor, where interactions are described as a global scenario. The global scenario is then projected by a projection function into individual services. These services are composed in parallel within multiparty sessions to encode global interactions. The correctness of the projection is ensured by defining a global type for choreography interaction protocol. The projection function then projects this global type into local types which captures services' behaviours. Sessions are created by projecting session names in choreography specifications into

shared channels in services' specifications during the projection phase. As a result, services can communicate asynchronously if they share the same channel name. Also, in this calculus multicast and broadcast are not permitted.

Later in [28], a new version of the calculus in [87] has been proposed to facilitate communication in synchronous mode, and multicast mode. Multicasting is achieved by indicating more than one recipient for a message in the choreography specifications. The calculi in [28] also discuss the idea of service delegation where a service delegates its part in a session by passing its channel names belonging to this session to other services dynamically. This is a benefit of the mobility feature in the foundation calculus ($\pi$-calculus).

The mobility feature of the foundation calculus ($\pi$-calculus) has also been employed in [110] to facilitate compositions between choreographies, by passing session names for global interactions dynamically in messages.

Inspired by the work of session types calculi [53, 87], multiparty sessions are achieved in [92, 42] by projecting choreography scenarios into a set of orchestrated services. The work in [42] was further extended in [38] to achieve adaptable choreographies, i.e. a choreography scenario (protocol) which might contain dynamic scopes that represent a part of a code which will be allocated dynamically. The correctness of adaptable choreographies is ensured by using a variation of the contracts language of [42], where services' contracts are proposed to govern the projection correctness between the global protocol and the orchestrated services. Originally, this notion of compliance was proposed in [55, 41] to ensure the compliance between client requirements and service capabilities (their contracts) while searching for services in the service discovery phase of SOC.

### 2.1.2 Mobility in SOC calculi

One of the significant features of SOC systems is their ability to reconfigure themselves dynamically. Therefore, any process calculus proposed for modelling SOC systems should be equipped with primitives to facilitate dynamic reconfiguration; more specifically, primitives to model delegation of the communication capabilities from one service to another. In the literature, most of the process calculi proposed for SOC are based on the $\pi$-calculus. Mobility is achieved by allowing processes to

pass link names along current channels. If one process passes a link name to another process, then it delegates this communication link to the receiver, where the received name serves as a new channel. A set of rules has been set to govern name exchanges to avoid collisions.

As an extension to this mobility model, the whole process is allowed to migrate from one space to another. Ambient calculus [54] and Join calculus [70] admit this type of mobility. In addition, an extension to the $\pi$-calculus called Higher Order $\pi$-calculus [126] implements this type of mobility too. However, Ambient calculus is novel in introducing constructs to send a whole ambient (bounded place which contains computations, like processes and threads) or a subambient (nested ambient) as a movement of a whole computation domain instead of moving a single computation entity as processes.

Mobility was also studied within the standard CSP. Welch and Barnes present occam-$\pi$ [139] as a mobile version of occam (a concurrent programming language founded on CSP), and two different models for mobile CSP are suggested in [122, 124]. The first model introduces mobility into CSP by passing the right to use a channel instead of passing the channel itself. In the second model, a mobility model is suggested for Hoare's CSP [84] by passing channel names. The first model has been adopted by Vajar et al. [134] to achieve mobility in (CSP $\|$ B).

An interesting question is how mobility is employed within SOC calculi. As mentioned in the previous section $\pi$-calculus mobility is used to achieve service delegation in [28] and choreographies compositions in [110].

The idea of mobile sessions has been suggested as future work in [36] but no further studies conducted.

### 2.1.3 Mixed asynchronous/ synchronous communications in SOC calculi

The most well-known theory to reason about asynchronous communications is the asynchronous $\pi$-calculus ($\pi_a$) [86]. The $\pi_a$-calculus [127] is a variation of the $\pi$-calculus with no output guards. Instead, outputs are provided in the calculus as standalone processes. Asynchronicity is achieved by breaking the order of executing input/ output actions.

Adapting the concept of no-output-guards, an asynchronous version of an early CSP-based language was proposed in [37]. However, the parallel composition of the language in [37] is only allowed at the top-level (see [113] for more details).

Alternatively, buffers are a well known mechanism for the implementation of asynchronous communications, and they have been extensively used in process calculi literature.

Early proposals which support asynchronous communications by forcing interactions between two processes to always be mediated by buffers are described in [60, 29], and in [27] buffers have been introduced to the $\pi$-calculus to facilitate asynchronous communications as an alternative to the no-output-guards approach implemented in [86]. In [27], the encodability between the two calculi has been studied.

Furthermore, in the context of CSP, the use of buffered channels to facilitate asynchronous communications has been previously discussed by Hoare [84]. This model is not formally implemented, and assumes asynchronous communications only. Buffered channels within CSP have been also discussed in [124], where all or a set of channels can be selected to be buffered between two processes.

A variation of CSP calculus, CSP# (the input language of the PAT model checker [131]) also extends CSP's syntax with buffered channels to facilitate asynchronous communications in addition to the default synchronous communications, which permits mixed asynchronous/ synchronous communications.

In the context of SOC, to the best of our knowledge no mixed synchronous/ asynchronous communications are available yet. Calculi in Section 2.1.1 are either supporting synchronous communications like [35, 93, 36] or supporting asynchronous communications like [100, 87].

### 2.1.4 Transitionality and compensations

Compensations refer to the backward behaviour of the normal (forward) behaviour of a system. Compensations should be executed to recover the system to a safe state after failures.

Compensations can be encoded in any calculus as standard processes. However, a specific type of processes for compensations, usually named *compensation handlers*, is significant to distinguish compensations work flow from normal processes work flow

in order to facilitate reasoning on compensations.

The new type of processes can be installed in calculi statically or dynamically: statically, where the compensation handlers are known from the design time; or dynamically, where the compensation handlers are built dynamically during the run time by composing smaller compensations. The smaller compensations are associated to normal processes and form a new type of processes, namely *compensable processes*.

Basically, compensation handlers are used to recover systems to a consistent state after failures. The order in which compensations are recovered is determined by the calculus recovery mechanism. Recovery mechanisms in calculi can be classified as (adapted from [96]): *parallel recovery* where all processes compensate in parallel; *backward recovery* where parallel processes compensate in parallel and sequential processes compensate in reverse order; and *general dynamic recovery* where compensations can be updated and replaced while the system is running.

Among the first dynamic compensating calculi for concurrent applications is Structured Activity Compensation (StAC) [48, 46]. In fact, the first dynamic construction of compensation was introduced in SAGA [72] but SAGA is not a concurrent calculus.

The version of StAC in [48] needs explicit activation to start the compensations. Therefore, two proposals for improving compensating calculi were suggested in [49] and [46]: Butler et al. [49] introduce Compensating CSP (cCSP) by extending standard CSP with new primitives to build the compensation sequence from smaller compensations elements at systems run time; Bruni, Melgratti and Montanari [46] followed the same approach to introduce compensation to Parallel Sagas [72]. The main difference between these two works was in how the two calculi developed their formal semantics [43].

In the context of the $\pi$-calculus, several researchers studied the introduction of compensations into the $\pi$-calculus [109]. Firstly, Bocchi et al. [33] have developed Transactional $\pi$-calculus ($\pi t$-calculus), which extends the asynchronous $\pi$-calculus with a failure manager for the parallel recovery of statically installed compensations. After that, $Web\pi$ [99] has been proposed to refine the $\pi t$-calculus. In $Web\pi$ [99] compensations are installed statically and recovered in parallel without an explicit manager. Later, $Web\pi_\infty$ [105] was developed as an improved version of $Web\pi$ without time. According to [95], compensations for sequential transactions, in $Web\pi$ and $Web\pi_\infty$, are executed in parallel not in reverse order as expected. Dynamic compen-

sations were introduced to the $\pi$-calculus in $dc\pi$-calculus [135] with parallel recovery. General dynamic recovery was introduced to $\pi$-calculus in [95] then it was revised in [96]. Additionally, in [63] general dynamic recovery has been encoded in CCS.

Static handlers for compensations have been introduced in most SOC calculi such as COWS [100]. In addition, dynamic compensations have been introduced in a number of SOC calculi such as CC in [51] and SOCK in [79].

Recently, in the context of SOC calculi, reversibility is proposed to undo sessions' executions for debugging purposes [94, 73]. In addition, reversibility is suggested in [133] to facilitate sessions restarting or reverting to a particular point in case of errors, instead of aborting. Reversibility originated in [59, 116], and it represents backward behaviour as compensation. However, compensations were proposed to recover systems or sessions to a safe state after failures, whereas revisable actions are used to undo systems' or sessions' executions up to a particular point.

## 2.2 Previous works in Verification tools for SOC Process Calculi

Considering formal methods as modelling languages enables the formal verification of system properties, a feature not available in informal languages. However, formal languages are often perceived as hard to apply. Therefore, different tools were developed to facilitate the verification process.

Model checkers [25] have been used extensively to prove the correctness of software applications by searching the entire state space of the checked model to prove the intended properties. Searching the entire state space could raise the state explosion problem [141], where the number of states grows until it reaches the model checker's limit of memory. This could happen in the case of checking huge data or complex data with structures like lists or trees.

On the other hand, theorem provers or proof assistants are interactive software which are designed to help in deriving proofs for mathematical theorems [81]. However, theorem provers cannot be fully automated, so humans should be part of the loop to guide the prover to discover proofs. In other words, theorem provers are not push-button tools like model checkers. In addition, proofs could be very large

and span multiple files. On the positive side, theorem provers do not put a limit on the states' space and they are equipped with mathematical mechanisms to reason inductively over complex data structures.

In this thesis, we seek an automated reasoning tool which could be used to enhance the usability of our SOC calculus. Therefore, to achieve this purpose we prefer model checkers rather than theorem provers.

Examples of well-known general model checkers are: SPIN [85] model checker which accepts models in Process Meta Language (Promela), a version of CSP process calculus; UPPAAl [101] model checker which verifies timed automata models; Maude [22] which uses rewriting techniques to check models; PAT [131] model checker which checks models written in C# language or CSP#, a variation of CSP process calculus; and FDR [12] model checker which is proposed as a refinement model checker for CSP models.

In the context of SOC calculi, CMC [26] is the model checker which supports SOC calculi by checking COWS models. However, in the literature, a number of model checkers like [64, 89, 71, 89, 112] are available to reason on SOC models using other formal methods like automata or Petri Nets. As stated previously we prefer process calculi over automata or Petri Nets due to the type of properties that can be checked by process calculi.

Additionally, in the context of web services, a number of model checkers like MCMAS [102], UMC [26], and ChorSLMC [26] are available which check agents models, UML4SOA models, and Choreography specifications respectively. However, these tools use different mechanisms therefore we cannot compare them to process calculi model checkers.

## 2.3 Motivation Example

We motivate our work by presenting a simpler version of the credit request scenario of the finance case study [65, 66, 132], which we present in full in Chapter 7.

In the credit request scenario the client can request a credit by first logging into a bank portal service and provides his ID and his password. After logging in, the customer will be able to place a new request, that includes the desired amount and the securities which form a balance. The portal places the request with the balance

in the task list of clerks for processing. Later, a clerk will retrieve a task from the list and process it. If the balance is "notOK" then the request is rejected, and the client is notified accordingly. If the clerk's decision is to approve the credit then an offer is generated, and the customer can accept or reject the offer.

If an offer is generated and the client's decision is to accept the offer, then this offer is placed in the task list of an automated service which is notified to schedule the money transfer. Then the automated service will inform the client of the date of the transfer.

According to the description, this scenario has three main actors: the client, the bank portal, and the clerk. To design this scenario we start with the client establishing a session with the bank portal to access the bank services. We define a session here to avoid interference between the interactions of this client and other clients. In a session, all interactions are tagged with a key to distinguish them from other interaction in the system. Each session has its own unique key. In fact, if the client requests a credit then the client service will establish a multiparty session inside the bank to approve this request. Multiparty session is needed because the bank portal service will need the help from other services inside the bank system to complete the requested service. The suggested details of the services are as follows:

The client first logs into the bank portal by providing his credentials (userID and password). More specifically, the **client** establishes a session with **BankPortal** service using soaCSP invocation primitive ($BankPortal \Leftarrow \{\}$), then sends a message **login** containing the relevant information, then waits for either one of two messages: (**Valid**) if his credentials match an existing user, or (**notValid**) with an error message if not. If he receives the **Valid** message then he can start using the bank services. If the logging in failed then the client will be notified by the message **notValid.ErroMSG**, and the session will be closed.

If his credentials are **Valid**, then the client can request a credit by using (invoking) **CreditRequest** service. The new invoked service (**CreditRequest**) will join the current session by using the soaCSP primitive ($CreditRequest \Leftarrow^{+} \{\}$) instead of creating a new session using the primitive ($\Leftarrow \{\}$), therefore, events in this service will be tagged with the same key of the current session instead of generating a new key for the new session. As a result, if a service joins a running session then it can directly communicate with all services inside this session without needing any intermediate

messages to circulate data between sessions. As soon as **CreditRequest** service joins the session the requesting process can start by sending a message **request** containing the desired amount and the securities, which forms the balance. The client waits afterwards for either one of two actions that inform of the bank's decision: either he receives an offer which he can accept or reject, or he receives a message (**requestDenied**) where the session will be closed. In the case in which the request is approved, the message **transferDate** is sent informing of the date when the funds are to be made available.

```
client= BankPortal ⇐ { login!ID!pass → ( (notValid.ErrorMSG → SKIP) □
        (Valid →  CreditRequest ⇐+ { request!amount!SEC →
            (userOffer □ (requestDenied → SKIP)) } ) ) }
userOffer= offer?amount → ( ( AcceptOffer → transferDate?date → SKIP)
                ⊓ ( RejectOffer →  SKIP) )
```

Services in the bank site need to be available always, so when they are executed once they will not disappear from the system. In soaCSP, we use the notation $*$ with the service name to indicate that the service is persistently available, and we use the publishing primitive ($\Rightarrow$) to define a service.

In the bank side we suggest the following services to implement the credit request scenario:

Firstly, the **BankPortal** service receives the **login** message. After that, the service will search the full clients database to match the user ID and password with an existing record. We assume that this database is a shared database outside the service code, therefore we use $\diamond$ notation with the event name to indicate that this communication is evaluated outside the border of this session. The **client** will be notified accordingly.

```
*BankPortal ⇒ login?ID?pass → searchDB◇!ID!pass  → getDBresult◇?ser
        → if (ser==exists) then (Valid →  SKIP)
                    else (notValid.ErrorMSG → SKIP)
```

Secondly, the **CreditRequest** service starts by receiving the request from the client. If a new request is received, then a bank clerk will be asked to process the request and give a decision. The clerk can be consulted by placing the new request in

the clerks' task list which is managed by service **Clerklist**. Therefore, to consult a clerk we should first invoke the service **Clerklist** to join the running session. Later, the **Clerk** can retrieve the request and process it, then reply with his decision via message **assess.dec**; we explain how the clerk processes the request later in this section.

Given the decision, the **CreditRequest** service code acts accordingly: if the decision is "notOK" then the request is declined and the client is notified by message (**requestDenied**); otherwise an offer is generated indicating that the request has been approved.

```
*CreditRequest ⇒ request?amount?SEC  → ClerkList ⇐+
      { addtoClist!amount!SEC → assess?dec → if dec == notOK
        then (requestDenied → SKIP) else generateOffer }
```

We use the ordinary processes **generateOffer** to organise the code of the service **CreditRequest**. The process of generating an offer starts by sending the offered amount to the client. If the client accepts then a subsession is established with the service **TransferMoney** to schedule the date of transferring the money. If the client rejects the offer then the session is closed. Note that, we use ↑ notation with the event name (**transferDate**) to indicate that this message will be propagated to the upper session where the client service operates.

```
generateOffer= offer!amount → (( AcceptOffer →
       TransferMoney ⇐ { transMoney?amount → SKIP } )
                      □ ( RejectOffer →  SKIP) )
*TransferMoney ⇒ transMoney!amount → transferDate↑!date → SKIP
```

To implement the clerk's list we will use a buffer where requests from different clients are placed in this buffer then the clerk will retrieve requests, in a time suitable to him, one by one in order and process them, then return the decision to the respective client.

In soaCSP, we have three types of communications:

1. **Synchronous communications**: where two or more processes (or services) use the same message name and it is stated in the interface set of the parallel compositions that they should synchronise on it. Then, these processes will not

be able to evolve until all of them are ready to execute this message, and any data associated to this message and emitted by an outputting process will be delivered to the respective inputting processes.

2. **Interleaving communications**: where two or more processes (or services) have the same message name but this message is not included in the interface set of the parallel compositions. Therefore, the processes do not need to synchronise on it, and they are able to evolve independently of each other. However, any data associated to this message will be lost.

3. **Asynchronous communications**: Same as the interleaving communications but we use with them the special symbols for inputting and outputting ($!<, ?>$) instead of the standard CSP inputting and outputting symbols ($!, ?$). If the new symbols are used then a buffer will be attached to the message name, so any data associated to this message and emitted by an outputting process will be stored in this message buffer, then when the respective inputting processes are ready they will retrieve it from the buffer.

Most of the communications in this scenario are synchronous, however, in implementing the **ClerkList** service we will use asynchronous communications. The trick here is to use the buffer attached to the asynchronous communication to act as the clerk list. Moreover, asynchronous communications are preferable because the service is interacting with bank employees, and the bank employee can process requests in the list, in a suitable time for him, and this should not affect the work inside the service and the session.

As stated previously, the **ClerkList** service manages the clerks' list which we choose to be the message **Clist**. The service code starts with the service **CreditRequest** sending the message **addtoClist** and sending the relevant information. The service then stores the relevant information in the buffer of the message **Clist** by sending this information asynchronously via the message **Clist** to the clerk, i.e. $Clist \diamond !<$. We use the symbol $\diamond$ to indicate that this list should be shared between all clients' requests, not only the request in the current session.

However, to store the state information of the current session (i.e. the current session key), and to provide a way where the clerk can merge with the current session

to send his decision back, we send with the request information a mobile channel (*thisuser*). In soaCSP, when we communicate mobile channels (i.e. variable messages) outside session boundaries, these channels are sent along with the session key. Later, if the clerk retrieves the request and comes to a decision on this request, then, the clerk should send his decision back to the respective client by using the session state information which is stored with the mobile channel (i.e., *thisuser*). The sketch in Figure 2.1 graphically illustrates this service and its relevant processes.

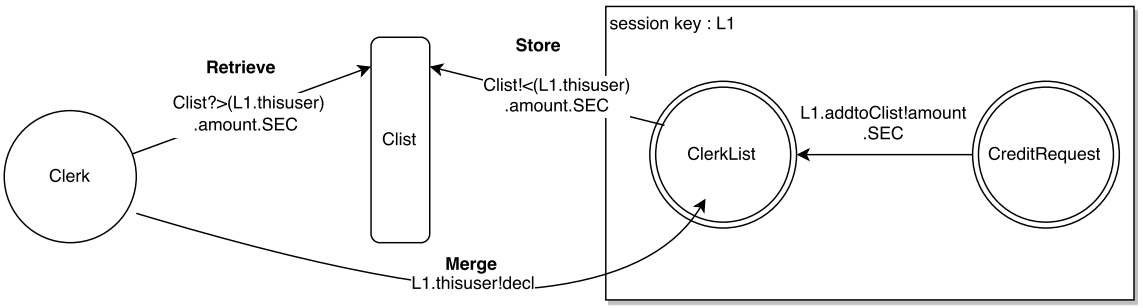


Figure 2.1: The finance case study messages sequence

The service **ClerkList** continues by sending back the decision to the **CreditRequest** service.

The clerk in this bank system is represented as an ordinary process; no need for creating a session, because the state information for clients is already stored with the request in the clerk list. The details of how the bank clerk processes the request are not relevant for this example.

```
*ClerkList ⇒ addtoClist?amount?SEC → Clist◇!<(thisuser).amount.SEC →
    thisuser?>dec → assess!dec → SKIP
Clerk= Clist?>(x).amount.SEC → processC  → x!<dec  → Clerk
```

All theses processes and services are working in parallel in the following system:

```
system = client ||A ( *BankPortal ||| (
                *CreditRequest ||B (*TransferMoney ||| *ClerkList) ) )

Where
A={| login, Valid, notValid, request, requestDenied, offer, AcceptOffer,
```

```
       RejectOffer, transferDate |}
B={| transMoney ,addtoClist, assess |}
```

$$\texttt{finance = (system ||| Clerks)} \parallel_{\{|\ a\leftarrow, a\rightarrow\ |\ a\in\ \Sigma\ |\}} \texttt{B}_{\Sigma}$$

As can be seen from the previous example, in designing soaCSP we were very careful to install sessions and retain the programming style of the process calculi. In other words, we govern the communications in soaCSP with session keys whenever needed and leave it free otherwise. This will reduce the need for interface messages and intermediate services to integrate ordinary processes which come usually from legacy systems, like the clerk or the shared database in our example. However, soaCSP provides a mechanism for processes to communicate with services as shown in the example.

Moreover, soaCSP provides a flexible communication model where designers can choose between synchronous, asynchronous, or interleaving mode based on system requirements. For instance, in our example the online bank system should return immediate response to clients, therefore, we choose synchronous communications for bank portal. On the other hand, consulting a clerk for approving the client request is done asynchronously as the clerk specification sates that the clerk can process client request in a time suitable for him.

In soaCSP, sessions can include more than two services (multiparty). Multiparty sessions have been implemented previously in several proposals like the Conversation Calculus (CC) [137, 136] and $\mu$se [45], which are multiparty orchestration calculi. However, the novelty in soaCSP is that services inside a session can interact in a multiway style, e.g. one service sends a message and several services receive this message immediately (at the same time). This feature is not shown in this example but can be seen in the case study in Chapter 7 page 176.

Moreover, sessions in soaCSP are simply created by invocations to service names. soaCSP does not require defining other modules, like parties and conversation contexts in CC or sites in $\mu$se. For instance, a similar scenario can be implemented in CC as follows (adopted from [136]):

```
Client ◄ [
     new BankSystem.BankPortal ⇐
```

```
        login↓!(ID,pass). (notValid↓?(ErrorMSG)  +
                          (Valid↓?().
                           join BankSystem.CreditRequest ⇐
                             request↓!(amount,SEC). (requestApproved↓?().
                                 transferDate↓?(date) +
                                              requestDenied↓?() ) ) ) ]
BankSystem ◄ [
     *def BankPortal ⇐
         login↓?(ID,pass).
           join DB.SearchUser ⇐
             searchDB↓!(ID,pass).getDBresult↓?ser.
               if ser = exists then notValid↓!(ErrorMSG)
                 else Valid↓?().
               |
     *def CreditRequest ⇐
          request↓?(amount,SEC).
              this(clientChat).  new Clerk.ClerkList ⇐
                                   addtoClist↓!(clientChat,amount,SEC).
                                   assess↓?(dec).
               if dec = notOK then requestDenied↓!()
                                    else (requestApproved↓!().
                                    transferDate↓!(date) ]
Clerk ◄ [
     *def ClerkList ⇐
        addtoClist↓?(clientChat,amount,SEC).
         join BankEMP.ClerkEMP ⇐
               Savelist↓!(clientChat,amount,SEC).
               processed↓?(clientChat,dec).
                clientChat ◄ [ assess↓!(dec) ] ]
```

As can be seen in the CC code (for CC details refer to [137, 136]), we need to define services to communicate with shared databases and clerks, and for extra messages to circulate data between involved services. Additionally, in CC we need to define parties (contexts) like BankSystem, Clerk, and Client, which act as holders for CC code and service definitions like BankPortal and CreditRequest, and in the same time they act

as conversation access points to establish multiparty sessions. This extra work is not required in soaCSP as previously explained.

Also, CC does not support complex data, we could not provide extra details when it came to databases or lists, e.g. only an interface to the clerk service can be provided.

While session terminations are noticed in soaCSP using the termination primitve *SKIP*, session terminations have not been discussed in CC.

CC relies on the $\pi$-calculus communication model, therefore, communications in CC are synchronous in general, whereas in soaCSP it can be synchronous, asynchronous, or interleaving.

Although CC relies on the $\pi$-calculus communication model, mobile communications are prohibited, and session state information can be saved and sent in CC by using the new primitive *this*, not via mobile channels. To use this distinction feature of CC a designer should capture session state information (i.e. conversation contexts) in a variable and send it to the respective destination. Whereas in soaCSP state information is sent along mobile communications without further intervention from the designer.

A similar scenario to this example, has also been implemented previously in BPEL [66]. As can be seen from a comparison between soaCSP and BPEL codes for this scenario that soaCSP script is much more concise. In BPEL a number of services and messages have been added to circulate data between main services and to provide interfaces for processes to allow them to work under BPEL session systems. Additionally, we point out that BPEL sessions are between two entities only, which means that we need extra messages to circulate data between services in different sessions and we can not guarantee immediate response from more than one service at a time.

It has been argued by the authors of Service Centred Calculus (SCC) [35] that general purpose concurrent calculi like $\pi$-calculus [127] are not suitable for SOC communications, since the different communication patterns are mixed, and most of the interesting properties like sessions are not directly reflected in the calculus scripts. The argument is supported with an encoding of the SCC calculus into the $\pi$-calculus, where we can see that all the information pertaining to sessions, invocations and publishings get mixed up with the other communication primitives, which makes it difficult to reason on the resulting process.

Comparing soaCSP to SCC or to the Stream-based Service-Centered Calculus

(SSCC) [93], which accommodates the same session mechanism of SCC, highlights a main point which is sessions in SCC and SSCC are two-party whereas in soaCSP sessions are multiparty.

## 2.4 Preliminary Theories

In this section we recall the preliminary definitions and theories which are directly related to our work; expert reader can skip this section and go directly to the next chapter.

### 2.4.1 Labelled Transition Systems

As stated in the Introduction chapter we choose to use operational semantics to describe the behaviour of our process calculus. Therefore, in this section we provide a brief introduction to operational semantics.

Operational semantics describes the different constructs of a language by showing their computation steps [68]. Structured Operational Semantics (SOS), was introduced by Plotkin in [117] as a logical method to define operational semantics.

The basic idea behind SOS is to define the behaviour of a language by showing the computation steps of its constructs. This will provide a structural, i.e. syntax oriented and inductive, view on operational semantics. Constructs' computation steps are defined as a set of transition relations, which take the form of a set of inference rules. Inference rules define the valid transitions of a syntax construct in terms of the transitions of its components.

SOS generates a Labelled Transition System (LTS). Labelled transition systems (LTS) is a well-known formal model underlining language semantics. In LTS, a system is represented with a set of states (called configurations). These states are closed terms over an algebraic signature. Transition from one state to another is made by firing an action in the form of a labelled transition with the name of this action.

Formally speaking, LTS is defined as a tuple ($\mathit{configs}$, $\mathit{labels}$, $\longrightarrow$) where:

- $\mathit{configs}$ is a set of configurations, where configurations are the system states along with any environmental variables or stores.

- *labels* is a set of labels, where labels are the actions which a language construct is ready to perform.

- $\longrightarrow \subseteq$ *configs* $\times$ *labels* $\times$ *configs* is a ternary relation. If $p, q \in$ *configs* and $a \in$ *labels* then the tuple $(p, a, q) \in \longrightarrow$ can be written as $p \xrightarrow{a} q$.

**Extensional equivalences for transition systems**

These theories are used to study equivalences between Transitions Systems. This is achieved through a set of relations, however, before stating these relations, following [127], we define below weak transitions, assuming $p, p'$ are configurations in a LTS:

**Definition 2.1** (Weak Transition)**.** 1. $\Longrightarrow$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$ (i.e. $\Longrightarrow$ is $\xrightarrow{\tau}{}^{*}$) usually called weak transition. Thus, $p \Longrightarrow p'$ expresses that $p$ can evolve to $p'$ by performing zero or more internal actions, denoted by $\tau$.

2. $\stackrel{a}{\Longrightarrow}$ denotes the relation $\Longrightarrow \xrightarrow{a} \Longrightarrow$, for $a \in \Sigma$. Thus, $p \stackrel{a}{\Longrightarrow} p'$ expresses that $p$ can evolve to $p'$ by performing the visible action $a$ with any number, possibly zero, of internal actions before and after $a$.

3. $\Longrightarrow^{*}$ denotes a sequence of 0 or more steps using $\Longrightarrow$ or $\stackrel{a}{\Longrightarrow}$.

We selectively present here the definitions of strong bisimulation, bisimilarity [127], and similarity [108] relations, which are used in conducting the proofs in this thesis, we assume $p, p'$ are configurations in a LTS, and $q, q'$ are configurations in another LTS.

**Definition 2.2** (Strong Bisimulation)**.** A symmetric relation $R$ is a **strong bisimulation** if whenever $p\,R\,q$, then: If $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ and $p'\,R\,q'$. Two processes $p$ and $q$ are strongly bisimilar if there is a strong bisimulation relation $R$ such that $(p, q) \in R$. We write $p \sim q$ if $p$ and $q$ are strongly bisimilar.

**Definition 2.3** (Bisimilarity)**.** The *bisimilarity* relation, denoted by $\approx$, is the largest symmetric relation such that whenever $p \approx q$, then:

1. If $p \xrightarrow{a} p'$, then $q \stackrel{a}{\Longrightarrow} q'$ and $p' \approx q'$.

2. If $p \xrightarrow{\tau} p'$, then $q \Longrightarrow q'$ and $p' \approx q'$.

We say two processes $p$ and $q$ are bisimilar if $p \approx q$.

**Definition 2.4** (Similarity)**.** The *similarity* relation, denoted by $\preceq$, is the largest relation such that whenever $p \preceq q$, then:

1. If $p \xrightarrow{a} p'$, then $\exists q'.\, q \overset{a}{\Longrightarrow} q'$ and $p' \preceq q'$.

2. If $p \xrightarrow{\tau} p'$, then $\exists q'.\, q \Longrightarrow q'$ and $p' \preceq q'$.

We say that process $q$ simulates process $p$ if $p \preceq q$.

### 2.4.2 Business Process Execution Language (BPEL)

The Business Process Execution Language (BPEL) is an XML-based executable language for specifying the work flow in service compositions. The first specification of BPEL was a joint effort by BEA, IBM, and Microsoft. This specification was improved and later announced as a standard of OASIS [14] in 2007.

Essentially, BPEL specifications are defined on top of the WSDL [15] interfaces to describe the behavioural aspects of a service. BPEL processes use WSDL interfaces to export and import interaction information. The Web Services Description Language (WSDL) is a W3C standard that is used to define services interfaces. The service interface is used to describe the functionality of a service by defining its structure of messages, and the signatures of its operations.

In the following section we present the syntax of BPEL. BPEL lacks formal semantics therefore an informal description is provided.

#### BPEL Syntax

BPEL language is an XML-based language. For that, the a BPEL document (specification) is a set of elements. BPEL, as in XML, uses the tag style to define elements. The tag starts with (< `element` >) and ends with (< /`element` >). Between these *angle brackets* are the element contents which consist of element attributes. These attributes are name/value pairs that set values which are needed to process this element. Finally, elements can have child-elements which can be defined in the same way.

The root element of BPEL language is the **process** element, i.e., < `process` > ... < /`process` >, which defines the behavioural model of a process activity (service composition). Generally speaking, a process element has three main parts: partners, where the services and client partners are defined; variables, where the exchanged messages are defined; and after that the structure of communications between this process and the partners is defined by means of control flow constructs. The control flow constructs are used to define the orchestration scenario of this process. Using BPEL language terminologies, these constructs can be a basic activity, or a structured activity which works as a container for basic activities and/or other structured activities. These parts are expressed as child element constructs in the `process` element as follows:

- **Partners:** by < `partnerLinks` > element which defines the set of services and client partners who can participate in this process activity. For each participant a role attribute should be set. If the participant is a client partner then the attribute `"myRole"` is used to define the role of this process if it has been invoked by that client. On the other hand, if the partner is another service which will be invoked by this process then the attribute `"partnerRole"` is used to define the role of this invoked service.

- **Variables:** by < `variable` > element. BPEL has a variable section where variables can be defined for use in the main activity. Variables are given names using the `name` attribute, and a type using one of the following attributes: `element` if it is a reference to an XML element; `type` if it is a simple XML type; and `messageType` if it is an input/output message, if it is a input/output message then this value of this attribute should match a message name in the partners. Typically, a variable should be defined for each input/output message that will be used in the activity. Variables can be assigned values via the < `assign` > construct in the main activity, and values of variables can be retrieved by using the function `getVariableData()`.

- **Control flow:** as stated above, the process orchestration can be defined as basic activities and structural activities. The basic activity includes the following:

  - < `receive` > element which indicates that this process service expects

information from the client partner. Here, the process acts as a service provider. Important attributes: `partnerLink` is used to set the name of the client partner; `variable` is used to set the name of the variable where the incoming request message should be stored; and if the attribute `createInstance` is set to `"yes"`, then this service is a persistent service and on the receiving of the message an instance of this service will be created. Information of each instance is maintained through the state of a special set of variables, namely the correlation set. Correlation sets are defined by using the element < `correlationSets` >.

- < `reply` > element which is used if the information received in element < `receive` > needs a response. Important attributes: `partnerLink` is used to set the name of the partner; `variable` is used to set the name of the variable that holds the returned message.
- < `invoke` > element which indicates that this service invokes an operation from one of the services partners. Here, the process acts as client. Important attributes: `partnerLink` is used to set the name of the service partner; `inputVariable` is used to set the name of the variable where the input message that will be used to communicate with the service partner is stored. `outputVariable` is used to set the name of the variable where the returned message is stored if a response is needed.
- < `assign` > element which is used to update the value of a variable with a new value. Within < `assign` > element, elements < `copy` >, < `from` >, and < `to` >, can be used to copy messages and variable values from one variable to another.
- < `validate` > element which is used to validate the value of a variable.
- < `wait` > element which delays the execution of the current activity for a period of time. Its value can be a period of time or a predefined date.
- < `exit` > element which immediately terminates the current instance of the service.
- < `empty` > element which does nothing.
- < `throw` > element which signals an internal fault.

- < `rethrow` > rethrow a fault which was caught by the immediate enclosing fault handler.
- < `extensionActivity` > element which extends the current activity with a new activity which is not in the process definition.

The structural activities include the following:

- < `sequence` > element in which child elements are executed in a sequential order.
- < `flow` > element in which child elements are executed in parallel. Dependencies, i.e., synchronization relationships, between parallel activities are defined through element < `link` > which is a conditional transition that connects source activity with a target activity.
- < `if` >, < `else` > and < `switch` > elements which implement the standard conditional branching statements *if-then-else* and *switch*. < `elseif` > is an optional element within the < `if` > element to define a second level of *if*.
- < `forEach` >, < `while` > and < `repeatUntil` > elements which implement the standard loop statements *for*, *while* and *repeat-until*.
- < `pick` > element which waits for the occurrence of one event from a set of events, then executes the activity associated with that event.
- < `scope` > element which encapsulates a part of the process logic into a scope.

In addition to activities, BPEL uses four different handlers to handle termination and fault signals announced in a scope as follows:

- < `faultHandler` > is used to describe what to do in case of errors.
- < `compensationHandler` > is used to describe the roll-back activity for any successfully terminated activity within the scope where the error is occurred. Compensation handlers could be triggered by < `compensate` > or < `compensateScope` >.

- < `terminationHandler` > is used to describe what to do in case the scope is forced to terminate.
- < `eventHandler` > is used to describe what to do in case a predefined event occurs. The predefined event can be the arrival of a message or a timeout alarm.

As previously mentioned, BPEL process definition relies on XML Schema and WSDL for the definition of datatypes and service interfaces. Therefore, BPEL has a number of operations dealing with XML data and XML Schema and their parsing which is out of the scope of this thesis. For instance, in this thesis we do not consider elements like < `import` > which is used to to declare a dependency on external XML Schema or WSDL definitions.

### 2.4.3 The calculus of Communicating Sequential Processes (CSP)

CSP was developed as a modelling language for concurrent systems (hardware or software) [84]. Concurrent systems are systems where entities communicate and exchange data while they are running in parallel. In CSP, entities are represented as processes. An atomic process is a set of events which it can perform. These events are sequentially composed (prefixed) in a way which reflects the process's behaviour. In turn, different processes can be composed to construct a system. In CSP, processes can be composed using a set of operators which includes: choice, sequential, synchronous parallel and asynchronous parallel (interleaving).

The version of CSP presented in this section is usually considered as the standard CSP [121, 124]. Standard CSP is slightly different from the original Hoare's CSP [84]. In this thesis, the differences will be explicitly indicated, otherwise, standard CSP is equivalent to Hoare's CSP.

Events in CSP can have several forms as follows:

- **simple events:** a single literal name which describes the action to be performed.
- **compound events:** names can be made compound by using the *dot* (.). A literal name can be combined with another literal to form compound names like

*right.street*. Literal names can be combined with variables to assign an index to names, e.g. *message.x* where $x$ is an integer, this can be instantiated by *message.1*. There is no limit to the use of (.) to make compound names. For instance, *right.street.x.y* is a valid name.

- **communication events (channels):** The (.) can be replaced by (!,?) to give the sense of the direction of the channel. The symbol (!) indicates that a process is passing data along this channel. For instance, $m!3$ is a channel $m$ passing the value 3 to a parallel process. The symbol (?) indicates that a process receives data along this channel from a parallel process. For instance, $m?x$ is a channel $m$ receiving a value in the variable $x$. A type definition can be given to the variable $x$. For instance, $m?x{:}\mathbb{N}$ is a channel $m$ receiving an *integer* value in the variable $x$. CSP admits multipart events where input and output through the same channel are interleaved. For instance, $m?x{:}\mathbb{N}!y$ represents a channel $m$ receiving an *integer* value in the variable $x$ and passing the value of variable $y$. CSP forces multiway handshaken communications where all participants should agree to perform a synchronized event.

Basically, a CSP process can be defined by four behavioural descriptions [121, 124]:

- **Trace:** is a finite sequence of observable events which a process can perform.
- **Refusals:** is a set of events which a process may refuse to perform.
- **Failure:** is a set of events that a process may refuse to perform after a particular trace. Failure is represented as a pair of trace and refusal.
- **Divergence:** is a trace which after it, the process behaviour is not defined.

In the theory of CSP these behavioural descriptions are used to define the denotational semantics of CSP. *Denotational models* are mathematical models which give a meaning to a language's constructs by describing the effect of these constructs [68]. Although in this thesis we develop an operational semantics of our calculus based on the operational semantics of CSP as mentioned in the Introduction chapter, in the following we explain briefly CSP denotational models. This is because, CSP denotational models are used in Chapter 7 and mentioned in Chapter 9. Moreover, CSP denotational semantics is the first and distinctive semantics of CSP.

Let $p$ be a CSP process and $\Sigma$ denotes the universal sets of all events in a CSP model, then the behaviour of $p$ can be defined by one of the following denotational models [121, 124]:

- **Traces model:** is the set of all possible traces of $p$. This set can be finite or infinite depending on the nature of the process, if it will terminate or not. (i.e. $traces(p) \subseteq \Sigma^*$, where $\Sigma^*$ denote the set of all sequences produced from $\Sigma$). Using this model we can reason on what a process can/ can not do (good traces/ bad traces).

- **Stable Failures model:** is the pair of traces and failures of $p$, which means it associates each trace in the trace model with the set of refusals after that trace (i.e. $(traces(p), failures(p))$, where $failures(p) = \{(s, A) | s \in traces(p) \wedge A \subseteq \Sigma\}$). Using this model we can reason on a range of system's properties including deadlock-freedom.

- **Failures/Divergences model:** is the pair of failures and divergences of $p$ (i.e. $(failures_{\perp}(p), divergences(p))$, where $failures_{\perp}(p)$ is an extended set of failures which can work with divergences, that is, $failures_{\perp}(p)$ includes the failures set of the process in addition to the set of pairs $(s, X)$ where $s \in divergences(p)$ and $X$ can be everything $failures_{\perp}(p) = failures(p) \cup \{(s, X) | s \in divergences(p)\}$). Using this model we can reason on a range of system's properties including deadlock-freedom and livelock-freedom.

In addition to the above denotational models CSP comes with a theory of refinements. Refinements are partial order relations between processes in a CSP model. If $p$ and $q$ are different processes in a CSP model, it can be said that $p$ refines $q$ if every behaviour of $p$ is also a behaviour of $q$. If $p$ refines $q$ and $q$ refines $p$ then $p$ is equivalent to $q$. Refinements are a distinctive feature of CSP. In CSP there are three types of refinements [121, 124]:

- **Traces refinement:** A process $q$ is a traces refinement ($\sqsubseteq_T$) of $p$, if all the possible traces of $q$ are also possible for $p$, i.e. $p \sqsubseteq_T q$ if and only if $traces(p) \supseteq traces(q)$.

- **Failures refinement:** $q$ is a failures refinement ($\sqsubseteq_F$) of $p$, if all the possible failures of $q$ are also possible for $p$, i.e. $p \sqsubseteq_F q$ if and only if $traces(p) \supseteq traces(q) \wedge failures(p) \supseteq failures(q)$.
- **Failures/Divergences refinement:** $q$ is a failures/divergence refinement ($\sqsubseteq_{FD}$) of $p$, if all possible failures of $q$ is also possible for $p$ and all possible divergences of $q$ is also possible for $p$. $p \sqsubseteq_{FD} q$ if and only if $divergences(p) \supseteq divergences(q) \wedge failures_{\perp}(p) \supseteq failures_{\perp}(q)$.

As aforementioned, we develop the operational semantics of soaCSP based on the operational semantics of CSP. For that, we devote the rest of this section to explain the operational semantics of CSP in detail.

### CSP Semantics

In this section, we present CSP syntax and CSP operational semantics. We assume the notations in Notations page (i.e. x).

CSP processes are defined by the following grammar.

$$\begin{aligned} p, q ::= \quad & a \rightarrow p \mid p \Box q \mid p \sqcap q \mid p \|_A q \mid p; q \mid p \backslash A \mid p[\![R]\!] \mid \mu p.f(p) \\ & \mid p|||q \mid SKIP \mid STOP \end{aligned}$$

where $a$ can be the name of an atomic action, inputting through channel $a$ (written as $a?x$), outputting through channel $a$ (written as $a!x$), or a combination of them (e.g. $a!x?y$). The interleaving ($|||$) and the parallel composition ($\|_A$) operators have indexed versions written as $|||_{i=1}^{N}$ and $\|_{A_{i=1}}^{N}$ respectively. Note that, in CSP, the parallel composition ($\|_A$) is called generalised parallel composition. However, CSP has another version, called alphabetised parallel composition [124] and written as: $p \,{}_{\alpha p}\|_{\alpha q}\, q$, where participants synchronise in all shared events. The alphabetised parallel composition can be encoded in the generalised parallel composition by setting the interface set to be $(\alpha p \cap \alpha q)$. Here, $\alpha p$ denotes $p$'s alphabet which is the set of events that the process $p$ can perform. From now on when we write parallel composition we mean the generalised parallel composition.

The operational semantics of CSP is given in Figure 2.2, where $\xrightarrow{a}$, $\xrightarrow{\tau}$, $\xrightarrow{\surd}$ denote a labelled transition relation. In the semantics, $a$ can be an atomic action,

$$\text{(skip)}\ \frac{}{SKIP \xrightarrow{\surd} STOP} \quad \text{(prefix)}\ \frac{}{(a \to p) \xrightarrow{a} p}\ a \in \Sigma$$

$$\text{(exch1)}\ \frac{p \xrightarrow{a} p'}{p \,\Box\, q \xrightarrow{a} p'} \quad \text{(exch2)}\ \frac{q \xrightarrow{b} q'}{p \,\Box\, q \xrightarrow{b} q'}\ a, b \in \Sigma^{\surd}$$

$$\text{(exch3)}\ \frac{p \xrightarrow{\tau} p'}{p \,\Box\, q \xrightarrow{\tau} p' \,\Box\, q} \quad \text{(exch4)}\ \frac{q \xrightarrow{\tau} q'}{p \,\Box\, q \xrightarrow{\tau} p \,\Box\, q'}$$

$$\text{(rec)}\ \frac{}{\mu p.f(p) \xrightarrow{\tau} f[\mu p.f(p)/p]} \quad \text{(seq1)}\ \frac{p \xrightarrow{a} p'}{p\,;\,q \xrightarrow{a} p'\,;\,q}\ a \in \Sigma^{\tau}$$

$$\text{(seq2)}\ \frac{p \xrightarrow{\surd} p'}{p\,;\,q \xrightarrow{\tau} q} \quad \text{(inch1)}\ \frac{}{p \sqcap q \xrightarrow{\tau} p} \quad \text{(inch2)}\ \frac{}{p \sqcap q \xrightarrow{\tau} q}$$

$$\text{(hid1)}\ \frac{p \xrightarrow{a} p'}{p \setminus A \xrightarrow{a} p' \setminus A}\ a \notin A \quad \text{(hid2)}\ \frac{p \xrightarrow{a} p'}{p \setminus A \xrightarrow{\tau} p' \setminus A}\ a \in A$$

$$\text{(hid3)}\ \frac{p \xrightarrow{\surd} STOP}{p \setminus A \xrightarrow{\surd} STOP} \quad \text{(rem1)}\ \frac{p \xrightarrow{a} p'}{p[\![R]\!] \xrightarrow{b} p'[\![R]\!]}\ a\,R\,b$$

$$\text{(rem2)}\ \frac{p \xrightarrow{\tau} p'}{p[\![R]\!] \xrightarrow{\tau} p'[\![R]\!]} \quad \text{(rem3)}\ \frac{p \xrightarrow{\surd} STOP}{p[\![R]\!] \xrightarrow{\surd} STOP}$$

$$\text{(intv1)}\ \frac{p \xrightarrow{a} p'}{p \,|||\, q \xrightarrow{a} p' \,|||\, q}\ a \in \Sigma^{\tau\surd} \quad \text{(intv2)}\ \frac{q \xrightarrow{a} q'}{p \,|||\, q \xrightarrow{a} p \,|||\, q'}\ a \in \Sigma^{\tau\surd}$$

$$\text{(par1)}\ \frac{p \xrightarrow{a} p'}{p \ \|_A\ q \xrightarrow{a} p' \ \|_A\ q}\ a \notin A \quad \text{(par2)}\ \frac{q \xrightarrow{a} q'}{p \ \|_A\ q \xrightarrow{a} p \ \|_A\ q'}\ a \notin A$$

$$\text{(par3)}\ \frac{p \xrightarrow{a} p' \qquad q \xrightarrow{a} q'}{p \ \|_A\ q \xrightarrow{a} p' \ \|_A\ q'}\ a \in A \quad \text{(parT1)}\ \frac{p \xrightarrow{\surd} STOP}{p \ \|_A\ q \xrightarrow{\tau} \Omega \ \|_A\ q}$$

$$\text{(parT2)}\ \frac{q \xrightarrow{\surd} STOP}{p \ \|_A\ q \xrightarrow{\tau} p \ \|_A\ \Omega} \quad \text{(parT3)}\ \frac{}{\Omega \ \|_A\ \Omega \xrightarrow{\surd} STOP}$$

Figure 2.2: CSP's operational semantics

or $a.x$ (where $x$ can be variable or data) to represent input/output events through channel $a$. We explain below the main constructs in CSP and their semantics (see

Figure 2.2).

The process *STOP* is the process which does nothing. The process *SKIP* is the process that immediately terminates successfully. *Prefixing* $(a \rightarrow p)$ is the process which is ready to engage in event $a$ and then behave as $p$. *External choice* $(p \Box q)$ is the choice process which is resolved by the environment offering the first action of $p$ or $q$. *Internal choice* $(p \sqcap q)$ is the choice process which is resolved internally; thus, either alternative can be available after an internal action $\tau$. *Recursive functions* $(\mu p.f(p))$ are defined using an explicit fixed point notation, where $f(p)$ is a CSP term involving process $p$, and $\mu p.f(p)$ defines exactly the same process as $p = f(p)$. The law of recursion here is that the recursively defined process $\mu p.f(p)$ satisfies the equation defining it, i.e., $\mu p.f(p) = f[\mu p.f(p)/p]$ (details of the fixed point recursion in CSP can be found in [124]). *Hiding* an action $a$ in process $p$ $(p \backslash a)$ prevents the external environment from observing it. If $R$ is a relation which maps events in set $A$ to the events in set $B$, then *renaming* $(p[\![R]\!])$ is mapping events from $A$ to $B$ in process $p$. In a *sequential composition* $(p; q)$, $q$ is executed if $p$ terminates successfully. In a *parallel composition* $(p \|_A q)$, $p$ and $q$ are executed in parallel and synchronise on events in $A$ only; events not in $A$ are interleaved. In a *parallel composition*, processes are not required to synchronise on the terminal event $\surd$. If $p$ or $q$ are ready to terminate successfully, i.e. evaluating the event $\surd$, then the process can terminate and evolve to an intermediate state $\Omega$ until both of the processes terminate, then the parallel composition will terminate successfully; this is called *distributed termination*. In *interleaving* $(p|||q)$ the events from processes $p$ and $q$ can be performed in any order.

In Figure 2.2 we provide CSP operational semantics as defined in [124]. However, this semantics should be extended to facilitate the description of the models and the proofs in this thesis. For that, we provide below the inference rules which extend the prefix rule to allow data to be communicated and to implement the behaviour of the *if-then-else* statement. CSP input/output prefix rules will be further discussed in Chapter 4. Moreover, we formally define below that if a process name appears in a script then this name should be replaced by its definition.

$$\text{(prefix-out)}\ \frac{}{(a!x \rightarrow p) \xrightarrow{a.x} p}\ a.x \in \Sigma \qquad \text{(prefix-in)}\ \frac{}{(a?x \rightarrow p) \xrightarrow{a.v} p[v/x]}\ a.v \in \Sigma$$

$$(\text{if1})\frac{b \xrightarrow{\tau} b'}{\texttt{if } b \texttt{ then } p \texttt{ else } q \xrightarrow{\tau} \texttt{if } b' \texttt{ then } p \texttt{ else } q}$$

$$(\text{if2})\frac{}{\texttt{if } TRUE \texttt{ then } p \texttt{ else } q \xrightarrow{\tau} p} \quad (\text{if3})\frac{}{\texttt{if } FALSE \texttt{ then } p \texttt{ else } q \xrightarrow{\tau} q}$$

where $b$ denotes a Boolean statement and it is evaluated according to the standard Boolean semantics.

$$(\text{def})\frac{}{N \xrightarrow{\tau} p}\, N = p$$

The standard functions on *lists* are widely used within CSP [84, 124] and are also used within this thesis. We provide below the syntax for list functions in CSP.

Let $ls$ be a list, then $null(ls)$ function checks if the received list is empty and returns a Boolean value ($TRUE$ or $FALSE$) indicating the result; $head(ls)$ function retrieves the first element in a list; and $tail(ls)$ function returns a list which contains all the elements in the received list except the first element.

Normal functions can also be defined in CSP. Normal functions can accept events, data, and processes as arguments. The purpose of functions is to apply one of the standard CSP operations on the arguments. For illustration consider the following example.

**Example 2.1.** The following example will define the function $nFun$ which checks the counter $i$, if it is equal to 10 then we execute the process which first sends the event $a$ then terminates with $STOP$, if not then execute the received process $p$.

$nFun(i, a, p) = \texttt{if } i > 10 \texttt{ then } (a \rightarrow STOP) \texttt{ else } p$

The standard list functions and normal functions are evaluated as $\tau$. However, the formal operational semantics for the standard list functions and normal functions are not provided in our sources [84, 124] and defining an operational semantics for functions in CSP is not the concern of this thesis. Therefore, we use the name (ListFun) to denote the execution of the standard list functions as $\tau$ and (Funs) to denote the execution of normal functions as $\tau$.

### 2.4.4 The $\pi$-calculus

In this section we recall the main concepts in the $\pi$-calculus. The $\pi$-calculus is a mobile calculus founded on the theory of CCS process calculus [107]. Mobility was

implemented in CCS by allowing processes to pass the names of links along the current channels [109]. If one process (namely "agent" in [109]) passes the name of a link to another process, then it delegates this communication link to the receiver, where the received name serves as a new channel.

Every process ($p$) in the $\pi$-calculus has a description of its behaviour which is defined by ($p \stackrel{\text{def}}{=} ...$). In addition, every process is associated with a set of names ($n(p)$) storing its current linkage network. The names set can be divided into two sets: the free names set ($fn(p)$) and the bound names set ($bn(p)$) where ($fn(p) \cup bn(p) = n(p)$). Free names represent public communication with other processes. Bound names represent private channels where communications are prohibited externally, but they can be used within the process itself.

Names in the calculus are by default free names. However, if they are enclosed between two brackets ($name$) then they are bound names. Bound names can appear in the calculus in two positions:

- **Positive prefix (input):** If $p$ is a process and $a(b)$ is an input prefix, then $a(b).p$ is the process which receives bound name $b$ through its free name $a$ then behaves as $p$. On the contrary, names in the negative prefix (output) as in $\bar{a}b.p$ are free unless otherwise stated.

- **Restriction:** If $p$ is a process then $c$ in $(\nu c)P$ is considered as private channel. In other words, $p$ can not use $c$ to communicate with the external environment. However, $c$ can be used to communicate within $p$ itself. Actions along private channels appear externally as $\tau$.

In such an environment where names can be sent in order to establish new communication links, the freshness of these new names raises a serious concern. Therefore, Milner *et al* in their paper [109] introduce the following set of rules to avoid collisions between names:

- If two names appear in two different scopes (even nested) then they are different.

- If one process passes a free name to another process then it will be free in the second process. If the process sends a bound name to another process then it will be bound in the second process. Note that, $bn$ and $fn$ sets of the second process grow silently.

- **Scope Intrusion:** If one process emits a name to another process and the second process contains the same name as a bound name, then the bound name should be $\alpha$-converted (renamed) to avoid a collision.
- **Scope extrusion:** If one process emits a bound name to another process then one of these situations can arise:
  - **Extending scope:** If the sender process is still using this name afterwards then the bound name's scope will be extended to include the receiver process.
  - **Migrating scope:** If the sender process is no longer using this name afterwards then the bound name's scope will be migrated to the receiver process.

  To illustrate the above cases consider the following example:

  **Example 2.2.** let $(\nu c)$ be a bound name which links $p$ and $q$. If $p$ passes $c$ to $f$, and $p$ still uses $c$ afterwards, then $c$ scope will be extended to include $f$. Thus, $c$ will be a shared private channel between $p$, $q$ and $f$. If $p$ will not use $c$ afterwards, then the role of $c$ will be changed from a private link between $p$ and $q$, to a private link between $q$ and $f$.

The rest of this section will be devoted to presenting the $\pi$-calculus syntax and operational semantics.

### The $\pi$-calculus Semantics

In this section, we present the $\pi$-calculus syntax and the $\pi$-calculus operational semantics.

Processes in the $\pi$-calculus are defined by the following grammar.

$$\pi ::= a(b) \mid \bar{a}b \mid \tau$$

$$p, q ::= \pi.p \mid p + q \mid (\nu c)\, p \mid p|q \mid !p \mid \mathbf{0}$$

The operational semantics for the $\pi$-calculus is given in Figure 2.3, where $\xrightarrow{\alpha}_{\pi}$ denotes a labelled transition relation, and here $\alpha$ denotes an input of the form $ac$,

$$\text{(OUT)}\ \frac{}{\bar{a}b.p \xrightarrow{\bar{a}b}_\pi p} \qquad \text{(INP)}\ \frac{}{a(b).p \xrightarrow{ac}_\pi p\{c/b\}} \qquad \text{(TAU)}\ \frac{}{\tau.p \xrightarrow{\tau}_\pi p}$$

$$\text{(SUM-L)}\ \frac{p \xrightarrow{\alpha}_\pi p'}{p + q \xrightarrow{\alpha}_\pi p'} \qquad \text{(PAR-L)}\ \frac{p \xrightarrow{\alpha}_\pi p'}{p \mid q \xrightarrow{\alpha}_\pi p' \mid q}\ bn(\alpha) \cap fn(q) = \emptyset$$

$$\text{(COMM-L)}\ \frac{p \xrightarrow{\bar{a}b}_\pi p' \qquad q \xrightarrow{ab}_\pi q'}{p \mid q \xrightarrow{\tau}_\pi p' \mid q'}$$

$$\text{(CLOSE-L)}\ \frac{p \xrightarrow{\bar{a}(c)}_\pi p' \qquad q \xrightarrow{ac}_\pi q'}{p \mid q \xrightarrow{\tau}_\pi (\nu\, c)\,(p' \mid q')}\ c \notin fn(q)$$

$$\text{(REP-ACT)}\ \frac{p \xrightarrow{\alpha}_\pi p'}{!p \xrightarrow{\alpha}_\pi p' \mid !p} \qquad \text{(REP-COMM)}\ \frac{p \xrightarrow{\bar{a}b}_\pi p' \qquad p \xrightarrow{ab}_\pi p''}{!p \xrightarrow{\tau}_\pi (p' \mid p'') \mid !p}$$

$$\text{(REP-CLOSE)}\ \frac{p \xrightarrow{\bar{a}(c)}_\pi p' \qquad p \xrightarrow{ac}_\pi p''}{!p \xrightarrow{\tau}_\pi ((\nu\, c)\,(p' \mid p'')) \mid !p}\ c \notin fn(p)$$

$$\text{(RES)}\ \frac{p \xrightarrow{\alpha}_\pi p'}{(\nu\, c)\, p \xrightarrow{\alpha}_\pi (\nu\, c)\, p'}\ c \notin n(\alpha) \qquad \text{(OPEN)}\ \frac{p \xrightarrow{\bar{a}c}_\pi p'}{(\nu\, c)\, p \xrightarrow{\bar{a}(c)}_\pi p'}\ c \neq a$$

Figure 2.3: $\pi$'s operational semantics

an output $\bar{a}b$ or $\bar{a}(b)$, or the silent action $\tau$. Below $bn(\_)$ denotes the function that computes the set of bound names, i.e. names under the scope of $\nu$ or input $a(\_)$, and $fn(\_)$ computes the set of free (not bound) names. We explain below the main constructs in Figure 2.3.

The process $\mathbf{0}$ is the terminated process. An *output* process ($\bar{a}b.p$) is ready to output the name $b$ and then behave as $p$. The process $\tau.p$ is ready to evaluate a internal action then behave as $p$. An *input* process ($a(b).p$) behaves, upon receiving $d$, as $p[d/b]$; we assume that $p$ is $\alpha$-renamed before applying $[d/b]$ if there is a name collision in the process. *New names* are defined using the binder $\nu$, that is, $(\nu c)\ p$, and it defines $c$ as a bound name in process $p$. *Sum choice* ($p + q$) is resolved by either $p$ or $q$ evaluating their first action. In the *parallel composition* ($p \mid q$), $p$ and $q$ are executed in parallel and synchronise on all shared events. *Replication* ($!p$) provides an unbounded number of copies of process $p$, i.e. $p \mid p \mid \ldots$

*Note.* The CSP alphabet symbol ($\alpha p$) coincides with the $\pi$ prefix symbol ($\alpha.p$). Nevertheless, we retain the notations from [124, 127] and it will be clear from the context which one is intended. In addition, we use actions or events to denote processes' atomic actions; action and events will be used interchangeably.

# 3

# Asynchronous CSP (CSPa)

## 3.1 Introduction

In the Service Oriented Computing (SOC) paradigm, services can communicate using messages solely, where messages can be sent synchronously or asynchronously. In the web services (WS) standards [14, 13], messages are sent according to predefined *interaction patterns* [67]. These interaction patterns include the *request-response* pattern and the *solicit-response* pattern, which represent synchronous communication. In synchronous communication, the initiator of the message suspends processing until it receives a response. On the other hand, the *one-way* and *notification* patterns represent asynchronous communication, where the initiator of the message continues running the next coded statement, without suspending processing.

Process calculi, such as SCC [35] and CaSPiS [36], provide a formal specification of interaction patterns with synchronous communications, allowing designers to reason about the correctness of SOC systems. Other formal systems, such as COWS [100] and Conversation Calculus [137], consider only asynchronous communications, since synchronous communications in the internet standards are usually implemented by network protocols such as TCP/IP [69], which are by default asynchronous. To the best of our knowledge, none of the formal calculi developed for SOC supports both synchronous and asynchronous communications.

In this chapter, we propose a new process calculus, called asynchronous CSP (CSPa), that supports mixed synchronous, interleaving and asynchronous communications. Our calculus can be described as a buffered version of the standard CSP [124].

CSP has been chosen as the foundation for our calculus because its communication model supports mixed synchronous/interleaving communications.

The novelty in CSPa is the introduction of *implicit* buffers, which are used in the channel semantics to facilitate asynchronous communications in a transparent way. In other words, CSPa includes asynchronous communication primitives, which rely on buffers, but designers do not need to create, maintain or terminate buffers. We provide an operational semantics that explains how buffers work.

Additionally, we study the relationship between our calculus and CSP, and we encode our calculus in the original CSP. Although the new constructs in CSPa do not enhance the expressiveness of CSP in the sense that the new constructs can be encoded in CSP, CSPa simplifies reasoning by replacing the encoded transitions with one transition.

**Contributions in this chapter**

1. We formally introduce direct primitives to facilitate asynchronous communication in CSP, by using built-in buffers in the semantics, where data can be stored and then retrieved.

2. Our model provides two kinds of asynchronicity: (i) Synchronous communication, but with a delay between inputting and outputting values. (ii) Interleaving communication, with values not lost but stored in the buffers.

3. We prove that the built-in buffers do not change the external behaviour of the system.

4. We study the relationship between CSPa and CSP, and we prove that the new constructs in CSPa can be encoded in CSP.

**Structure of this chapter** Section 3.2 introduces CSPa. Section 3.3 discusses the relationship between CSPa and CSP. Finally, Section 3.4 concludes the chapter and discusses related works.

## 3.2 CSPa Model and Semantics

In CSP, the parallel composition operator does not force events to be synchronised. Instead, it is parameterised by an interface set, which governs the synchronisation between participants. Events inside the interface set should be simultaneously evaluated, whereas events outside the set can be evaluated independently even if they are shared. Evaluating events independently does not pass channels' data from inputting to outputting processes. If designers want asynchronicity (i.e., data to be transmitted with delay), they need to define and maintain an explicit buffer. To avoid this burden, we include built-in buffers in CSPa, associated with events in $\Sigma$, and extend the semantics of CSP to model asynchronous communications. In this way, asynchronicity becomes a primitive notion in the calculus, at the same level as synchronicity, and designers do not need to deal with the implementation details of creating and maintaining buffers.

To implement this solution, we extend the channel syntax of CSP with two events: $a!{<}x$, which denotes adding data $x$ as the last element in $a$'s buffer ($B_a$), and $a?{>}x$, which denotes the consumption of the first element in $a$'s buffer ($B_a$). We also extend the transition relation as follows: The prefix $a?{>}x$ in a process indicates that this process is ready to accept any value of $a$'s type, i.e. the type of data that channel $a$ can accept. This input transition rule is similar to the input rule of CSP (see rule (prefix-in) in Section 2.4.3) except that the label here is $a{\leftarrow}.v$ instead of $a.v$. In general, the $a{\leftarrow}.x$ label can be read as "input $x$ into channel $a$" (inputting into a channel means outputting from the buffers).

$$\text{(asy-in)} \quad \frac{}{a?{>}x \rightarrow p \xrightarrow{a\leftarrow v} p[v/x]}$$

If the event $a!{<}x$ is used within a process, then this process is ready to send $x$. Here $x$ can be a value or a variable of $a$'s type. This output transition rule is similar to the output rule of CSP (see rule (prefix-out) in Section 2.4.3) except that the label here is $a{\rightarrow}.x$ instead of $a.x$. The $a{\rightarrow}.x$ can be read as "output the value $x$ from channel $a$" (outputting from a channel means inputting into the buffers).

$$\text{(asy-out)} \quad \frac{}{a!{<}x \rightarrow p \xrightarrow{a\rightarrow x} p}$$

The symbols $>$ and $<$ in these events denote a communication with $\mathrm{B}_a$, where $\mathrm{B}_a$ denotes the built-in buffer for channel $a$. $\mathrm{B}_a$ appears in the semantics but it is transparent for designers. $\mathrm{B}_a$ is formally defined as follows:

**Definition 3.1** (Event Buffer)**.** For each $a \in \Sigma$, an unbounded buffer with first come first out (FIFO) policy is defined, denoted by $B_a$, and implemented as a process parameterised by a list $s$, as shown below. We write $B_a(\langle\rangle)$ to represent an empty buffer, and we write $B_a(\langle x\rangle^\frown s)$ to represent a buffer containing an element $x$ followed by the elements in $s$. The operator $^\frown$ represents list concatenation.

$$
\begin{array}{lll}
B_a(s) = & \texttt{if} & \mathit{null}(s) \\
& \texttt{then} & \big((a?{<}x \rightarrow B_a(\langle x\rangle)) \quad \Box \quad \mathit{SKIP}\big) \\
& \texttt{else} & \big((a!{>}\mathit{head}(s) \rightarrow B_a(\mathit{tail}(s)) \quad \Box \quad a?{<}x \rightarrow B_a(s^\frown\langle x\rangle)) \\
& & \quad \Box \quad \mathit{SKIP}\big) \qquad \Box\ \mathit{SKIP}
\end{array}
$$

Recall that, $\mathit{null}(s)$, $\mathit{head}(s)$, and $\mathit{tail}(s)$ are standard functions on lists, which respectively check if a list is empty, retrieve the first element in a list, and return all elements except the first element in a list. The option *SKIP* will be discussed in Section 3.2.1.

In the definition of $\mathrm{B}_a$, $a!{>}x$ and $a?{<}x$ are the complementary events of $a?{>}x$ and $a!{<}x$ and are reserved for buffers only, that is, they cannot be used by designers. They are defined as follows:

$$\text{(buffer-in)} \quad \frac{}{a?{<}x \rightarrow p \xrightarrow{a \rightarrow v} p[v/x]}$$

where data is accepted from processes which evaluate $a!{<}x$. $a?{<}x$ and $a!{<}x$ will be synchronised using the original parallel composition of CSP.

$$\text{(buffer-out)} \quad \frac{}{a!{>}x \rightarrow p \xrightarrow{a \leftarrow x} p}$$

where data is consumed by processes which evaluate $a?{>}x$. $a!{>}x$ and $a?{>}x$ will be synchronised using the original parallel composition of CSP.

To allow these new events to be evaluated by the parallel composition, i.e. rules

(par1-par3) in Figure 2.2 page 42, and the other CSP operators in Figure 2.2, we create the set $\mathcal{B}$ to include the event types: $a \leftarrow .x$ and $a \rightarrow .x$ as follows:

$$\mathcal{B} = \{a\leftarrow, a\rightarrow |\ a \in\ channels(\Sigma)\}$$

where $channels(\Sigma)$ is the function which extracts the channel name from events.

After that, we update $\Sigma$ to become $\Sigma \cup \mathcal{B}$. Thus, the original operator of the CSP can now evaluate the new types of events.

According to Definition 3.1, the unbounded buffer has three options: (i) to input data unconditionally; (ii) to output data if the buffer is not empty; (iii) to terminate if the system terminates.

In designing $B_a$, FIFO policy is implemented by attaching new data to the end of the buffer list and processes always consume the first element in the list.

The intuition behind introducing implicit buffers is to introduce a delay between sending and receiving messages. This will allow processes to continue their execution without waiting for the receiver to get the message.

In two-way communications (between two processes), if buffers are introduced in the middle of a communication then the sent message from the first process will be stored in the buffer until the receiving process is ready to get the message. However, in multi-way communications (in which multiple processes communicate) as in CSP, we should explain what asynchronicity means. In our model, we want to retain the CSP model of multi-way communications with the addition of asynchronicity. In multi-way communications, buffered channels may introduce non-determinism, as Example 3.1 shows.

**Example 3.1.** Consider the system

$$a?>x \rightarrow SKIP \parallel a?>y \rightarrow SKIP \parallel a!<3 \rightarrow SKIP$$

where the processes are not synchronised (i.e. interleaving); we write $\parallel$ for parallel composition with an empty interface set.

According to our model the communications can happen in any order. If the output is done first then the value 3 will be stored in $B_a$ then retrieved by the input event. However, which input event (i.e. $a?>x$ or $a?>y$) will get the value from the

buffer is not specified. Either $a?>x$ gets the value and $a?>y$ waits for a new value, or vice versa.

In CSPa, we can avoid such non-determinism by synchronising between input events or output events and the buffers, as shown in the example below.

**Example 3.2.** Let the following communications take place in a system:

$a?>x \rightarrow SKIP \parallel_{\{|a?>|\}} a?>y \rightarrow SKIP \parallel a!<3 \rightarrow SKIP$

Assuming the buffer is empty, then according to our model the value 3 will be stored in $B_a$ then retrieved by the input event. Here, the input events are synchronised, therefore $a?>x$ and $a?>y$ should get the same value (which is 3).

Consequently, in CSPa, two kinds of asynchronicity are available:

1. Synchronous communication, but with a delay between inputting and outputting values. This can be achieved by synchronising on input events and output events. Thus, if buffers are empty, then all output events happen at the same time and upload data to buffers, then all input events get the same value from the buffers. Note that, CSPa adheres to CSP synchronisation rules, where the whole a.n event is considered in synchronisation. For instance, in CSP, an event $a.3$ can only be synchronised with $a.3$ or $a.x$ if $x$ is an input variable (in the latter case, $x$ will be substituted by 3).

   A less strictly synchronous communication can be achieved by synchronising on input events only (as Example 3.2) or on output events only.

2. Interleaving communication, with values not lost but stored in the buffers. However, in this case the order of execution of a buffer's events is non-deterministic, as Example 3.1 demonstrates.

The event buffers can be considered as an area of shared memory which can be accessed by all processes. We define below a process, which we call *buffered*-$\Sigma$, to be the parallel composition of all $\Sigma$'s event buffers.

**Definition 3.2** (Buffered-$\Sigma$)**.** The process *buffered*-$\Sigma$ (denoted by $B_\Sigma$) is the parallel composition of all the individual event buffers.

$$B_\Sigma = |||_{i \in \Sigma} \quad B_i$$

where the interleaving operators are used to emphasise that event buffers do not synchronise on inputting or outputting. $B_\Sigma$ evolves by performing transitions ($\Longrightarrow^*$) to new states: $B'_\Sigma, B''_\Sigma, \ldots$ (for the definition of $\Longrightarrow^*$ see Definition 2.1)

To allow the buffered-$\Sigma$ to work with the whole system we will install it in parallel with the system as a preprocessing step, thus allowing the execution of the system in asynchronous mode. The buffered-$\Sigma$ will work in parallel with the whole system and synchronise on all events which have $\rightarrow$ or $\leftarrow$. The buffered-$\Sigma$ is installed using a parallel composition operator due to the fact that communications between processes take place only if they are working in parallel.

**Definition 3.3** (Buffered System)**.** If $p$ is a CSPa process then the *buffered system* which can be built from $p$ and its associated buffered-$\Sigma$ is defined as follows:

$$p \| _{\{a\leftarrow, a\rightarrow | a\in\Sigma\}} B_\Sigma$$

We use the silent action $\tau$ to represent the preprocessing step where the system is placed in parallel with the buffered-$\Sigma$. This is safe as long as the externally visible behaviour of the system is unchanged by the addition of an extra starting state with a $\tau$ action, where this system has no option but to take this invisible action and behave like a buffered system. Additionally, this will conceal the effect of adding buffers to a system.

Finally, it is important to ensure that buffers do not introduce non-termination, i.e. if the system terminates then buffers should terminate as well.

To achieve this, we introduce a new termination method for CSPa processes, namely *synchronised termination*, which replaces *distributed termination* in CSP. In *synchronised termination*, CSPa processes synchronise on evaluating the event $\surd$, i.e. termination. Therefore, if processes terminate then the buffered-$\Sigma$ will be forced to terminate as well; see Proposition 3.1 for details (this feature is also significant in other parts of our calculus, to deal with compensations and sessions). The following rule implements the *synchronised termination*:

$$\text{(parST)}\ \frac{p \xrightarrow{\surd} STOP \quad q \xrightarrow{\surd} STOP}{p \parallel_A q \xrightarrow{\surd} STOP}$$

Note that, rule (parST) replaces rules (parT1,2,3) in Figure 2.2 page 42.

We devote the rest of this section to proving that our buffered system is simulated by the original system. That is, our buffered system does not introduce new external transitions which the original system cannot do.

**Theorem 3.1.** *For every $p \in CSPa$, $(p \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B'_\Sigma) \preceq p$, where $B'_\Sigma$ is any state of the process $B_\Sigma$ : $B_\Sigma \Longrightarrow^* B'_\Sigma$.*

*Proof.* To prove similarity according to Definition 2.4 we need to consider in turn the transitions performed by the buffered system and match them with the transition performed by process $p$.

According to the labelled transition system presented in Figure 2.2 page 42 and Section 2.4.3 page 41, then extended in Section 3.2, a process $p$ can do the following transitions:

- The buffered system performs $\tau$ as follows:

  $(p \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B'_\Sigma) \xrightarrow{\tau} (p \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B''_\Sigma)$ by rules: (rec) in Figure 2.2, (if1-3) in Section 2.4.3, (ListFun) in Section 2.4.3, and (par2) in Figure 2.2.

  The process $p$ does not change, therefore: $p \Longrightarrow p$

- The buffered process performs $a \leftarrow .x$ as follows:

  $(p \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B'_\Sigma) \xrightarrow{a\leftarrow.x} (p' \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B''_\Sigma)$ by rules: (asy-in), (buffer-out) in Section 3.2, and (exch1), (par3) in Figure 2.2.

  Then, this transition is matched by the process $p$ in the following way: $p \xrightarrow{a\leftarrow.x} p'$, by rule: (asy-in) in Section 3.2.

- The buffered system performs $a \rightarrow .x$ as follows:

  $(p \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B'_\Sigma) \xrightarrow{a\rightarrow.x} (p' \,\|_{\{a\leftarrow,a\rightarrow|a\in\Sigma\}}\, B''_\Sigma)$ by rules: (asy-out), (buffer-in) in Section 3.2, and (exch1), (par3) in Figure 2.2.

  Then, this transition is matched by the process $p$ in the following way: $p \xrightarrow{a\rightarrow.x} p'$, by rule: (asy-out) in Section 3.2.

- The buffered system performs $a$ as follows:

$(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{a} (p' \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma)$ by rules: (prefix), (par1) in Figure 2.2.

Then, this transition is matched by the process $p$ in the following way: $p \xrightarrow{a} p'$, by rule: (prefix) in Figure 2.2.

- The buffered system performs $a.x$ as follows:

  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{a.x} (p' \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma)$ by rules: (prefix-in) or (prefix-out) in Section 2.4.3, and (par1) in Figure 2.2.

  Then, this transition is matched by the process $p$ in the following way: $p \xrightarrow{a.x} p'$, by rule: (prefix-in) or (prefix-out) in Section 2.4.3.

- The buffered system performs $\surd$ as follows:

  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{\surd} STOP$ by rules: (exch1) in Figure 2.2 and (parST) in Section 3.2.

  This is because, according to Definition 3.1 and 3.2, buffered-$\Sigma$ should evaluate and synchronise on $\surd$ if the environment offers this event and terminates. In this case above, the process $p$ is the process who is offering the $\surd$ event.

  Then, this transition is matched by the process $p$ in the following way:
  $p \xrightarrow{\surd} STOP$.

□

### 3.2.1 Direction, deadlock, and termination

In CSP, the unit of data can be divided into several components (e.g. $ItemID.ItemQ$) and these components, in synchronous mode, can be simultaneously conveyed in both directions (e.g. $a?ItemID!ItemQ$, where $ItemID$ is received and $ItemQ$ is transmitted at the same time on channel $a$). In CSPa, similar to [84, 124], channels have a definite direction, where the whole unit of data is loaded or consumed in one direction at a time (e.g. $a?ItemID.ItemQ$, $a!ItemID.ItemQ$).

Adding buffers to a calculus can significantly affect the performance of systems and introduce extra design errors. For instance, a system may end up in deadlock if an output cannot be evaluated because a channel's buffer is full. To avoid introducing

deadlocks, Hoare suggests that these buffers be unbounded. We have followed Hoare's approach here.

Thanks to rule (parST), the buffered $\Sigma$, which has been installed in buffered-system (see Definition 3.3), will terminate if the system terminates. This is due to the termination option that is resolved externally by the environment.

**Definition 3.4** (Terminated Process)**.** Let $p$ be a CSPa process, then $p$ is a terminated process if it can not evolve more, and $p$ is terminating if there exists $p \xrightarrow{a_0} p_1 \xrightarrow{a_1} \ldots \xrightarrow{a_k} p_n \xrightarrow{a_n} p_{n+1} \nrightarrow$ where $n \geqslant 0$, and $\nrightarrow$ denotes that no further transitions are available.

If $a_n = \surd$ then this process successfully terminated, otherwise the process is blocked with no further transitions.

**Proposition 3.1.** *Buffers do not introduce deadlock or non-terminating sequences of transitions, that is: (i) If $B_\Sigma$ terminates successfully then p terminates successfully. (ii) If channel buffers are not empty when inputs are made, then the buffered-$\Sigma$ blocks only if p blocks. (iii) If $B_\Sigma$ has infinite steps of transitions then p also has them.*

*Proof.* We prove the proposition's cases as follows:

1. If the buffered-$\Sigma$ terminates successfully then the process $p$ also terminates successfully. This is because the $\surd$ events are observable in the environment if the process $p$ successfully terminates. Then, the $\surd$ event resolves the external choice with the *SKIP* option in channel buffers as Definition 3.1 shows, using rule (parST).

2. If channel buffers are not empty, that is the system will not block because there is no data to retrieve, then as a consequence of Theorem 3.1, if $p \| _{\{a\leftarrow, a\rightarrow | a\in\Sigma\}} B_\Sigma$ blocks or has infinite sequence of transitions, then $p$ has this block or infinite sequence of transitions.

□

## 3.3 The relationship between CSPa and CSP

In this section, we first discuss the encoding of CSPa into CSP, and then we reason on the correctness of our encoding by proving that the encoded processes are weakly

bisimilar to the original processes of our calculus (Theorem 3.2). This results show that CSPa does not strictly enhance the expressiveness of CSP in the sense that the new constructs in CSPa can be encoded in CSP. However, CSPa simplifies the specification and verification of asynchronous communications, in the context of CSP, by providing explicit primitives for modelling them.

Thanks to the encoding, we can use the denotational models of CSP to analyse the correctness of CSPa systems. Informally speaking, the main differences between CSPa and CSP are the hidden communications with the buffered-$\Sigma$ and the synchronised termination.

To facilitate the encoding we define the *fresh name term* as a termination signal which will be used to force synchronisation when processes terminate. This termination signal will be hidden later in the system using the *hiding* operator.

Moreover, we define the functions in-process-out-buffer ($iPoB$) and out-process-in-buffer $oPiB$ which change the event $a$ into $a \leftarrow$ and $a \rightarrow$ respectively. Thus, $iPoB(a)?x$ represents the events $a?{>}x$ and $a!{>}x$, and $oPiB(a)!x$ represents the events $a!{<}x$ and $a?{<}x$.

The functions $iPoB$ and $oPiB$ work as follows:

$$iPoB(a) = a \leftarrow$$

$$oPiB(a) = a \rightarrow$$

We also define the set $\Sigma_{comp}$ to be ($\Sigma_{comp} = \Sigma \cup \{term\} \cup \{a \leftarrow, a \rightarrow | a \in \Sigma\}$).

In addition, we encode $B_a$ (defined in Definition 3.1) to be the following CSP process:

$$\begin{array}{lll} B(a,s) = & \texttt{if} & null(s) \\ & \texttt{then} & \big((oPiB(a)?x \rightarrow B(a,\langle x\rangle) \quad \Box \quad (term \rightarrow SKIP)\big) \\ & \texttt{else} & \big((iPoB(a)!head(s) \rightarrow B(a,tail(s)) \,\Box\, oPiB(a)?x \rightarrow B(a,s^\frown\langle x\rangle)) \\ & \Box & (term \rightarrow SKIP)\big) \qquad \Box\, (term \rightarrow SKIP) \end{array}$$

Note that, we use $oPiB$ for inputting data into the buffer because we want to produce the same event to force the process and the buffer to synchronise. Bear in mind that,

an output event in the process is considered as an input event in the buffer.

We also encode $B_\Sigma$ to be the following CSP process: $BS = |||_{i \in \Sigma}\ B(i, \langle\rangle)$.

Finally, we define the function Asynchronous-System *AsynchSys* to encode the *buffered system* (see Definition 3.3) as follows:

$$AsynchSys(p) = p \parallel_{\Sigma_{comp}} BS$$

where $p$ is the encoded process.

The encoding from CSPa into CSP is defined as follows:

**Definition 3.5.** The encoding $[.] : CSPa \rightarrow CSP$ is defined as: $[.] = [\![p]\!] \setminus \{term\}$, where $[\![p]\!]$ is defined homomorphically except for the following:

$$\begin{aligned}
[\![SKIP]\!] &= term \rightarrow SKIP \\
[\![a?{>}x \rightarrow p]\!] &= iPoF(a)?x \rightarrow [\![p]\!] \\
[\![a!{<}x \rightarrow p]\!] &= oPiF(a)!x \rightarrow [\![p]\!] \\
[\![p \|_A q]\!] &= ([\![p]\!] \parallel_{\{term\} \cup A} [\![q]\!])
\end{aligned}$$

**Lemma 3.1.** *If $q = p \setminus A$ then:*

*i. if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Leftrightarrow q \xrightarrow{\sigma} q'$, where $q' = p' \setminus A$.*

*ii. if $\sigma \in A$ then $p \xrightarrow{\sigma} p' \Rightarrow q \xrightarrow{\tau} q'$, where $q' = p' \setminus A$. Also, $q \xrightarrow{\tau} q'$ implies either $p \xrightarrow{\sigma} p'$ for $\sigma \in A$ or $p \xrightarrow{\tau} p'$.*

*Proof.* To prove (i), we first prove that if $\sigma \notin A$ then $p \xrightarrow{\sigma} p'$ implies $q \xrightarrow{\sigma} q'$, where $q' = p' \setminus A$. If $\sigma \notin A$ then there are two cases:

1. If $\sigma$ is of the form $a, a{\leftarrow}, a{\rightarrow}$ or $\tau$, then according to rule (hid1) in Figure 2.2 page 42, if $p \xrightarrow{\sigma} p'$ and $\sigma \notin A$ then $q \xrightarrow{\sigma} q'$ and $q' = p' \setminus A$.

2. If $\sigma = \surd$, then according to rule (hid3) in Figure 2.2 if $p \xrightarrow{\surd} STOP$ then $q \xrightarrow{\surd} STOP$, which implies that $q' = STOP$. However, in CSP, $STOP \setminus A = STOP$ (see [124]), therefore $q' = STOP \setminus A$, as required.

Secondly, we prove that if $\sigma \notin A$ then $q \xrightarrow{\sigma} q'$ implies $p \xrightarrow{\sigma} p'$ where $q' = p'\backslash A$.

If $\sigma \notin A$ then there are two cases:

1. If $\sigma$ is of the form $a, a \leftarrow, a \rightarrow$ or $\tau$, then by rule (hid1) in Figure 2.2 if $q = p\backslash A \xrightarrow{\sigma} p'\backslash A$, this implies that $p \xrightarrow{\sigma} p'$.

2. If $\sigma = \surd$, then by rule (hid3) in Figure 2.2 if $p\backslash A \xrightarrow{\surd} STOP$, this implies that $p \xrightarrow{\surd} STOP$. However, in CSP, $STOP\backslash A = STOP$ (see [124]), therefore $q' = STOP\backslash A$.

Thus, if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Leftrightarrow q \xrightarrow{\sigma} q'$ where $q' = p'\backslash A$.

To prove (ii): first note that if $\sigma \in A$ then $p \xrightarrow{\sigma} p'$ implies $q \xrightarrow{\tau} q'$ and $q' = p'\backslash A$, as a direct application of rule (hid2) in Figure 2.2.

To complete the proof, we prove that $q \xrightarrow{\tau} q'$ implies $p \xrightarrow{\sigma} p'$, for $\sigma \in A$, where $q' = p'\backslash A$, or $p \xrightarrow{\tau} p'$ as follows:

$q \xrightarrow{\tau} q'$ can take place in two cases:

1. If $p \xrightarrow{\tau} p'$ by rule (hid1) in Figure 2.2.

2. If $p \xrightarrow{\sigma} p'$ and $\sigma \in A$ by rule (hid2) in Figure 2.2.

This concludes the proof. □

**Theorem 3.2.** *Let* $[.] : CSPa \rightarrow CSP$ *be the encoding in Definition 3.5. For every* $p \in CSPa$, $p \approx [p]$.

*Proof.* We give the proof only for the four non-homomorphic cases of the encoding. The homomorphic ones follow trivially.

We will show that the above four cases of the encoding are bisimilar to their source processes. These four cases are described as follows:

1. $p = SKIP$, $[p] = (term \rightarrow SKIP)\backslash\{term\}$

   The transition $SKIP \xrightarrow{\surd} STOP$ follows by rule (skip) in Figure 2.2 page 42, and is the only transition for $SKIP$.

   This transition is matched by the encoded process in the following way:

$[SKIP] \xrightarrow{\tau}\xrightarrow{\surd} STOP$ according to rules: (prefix), (hid2), and (skip) in Figure 2.2.

In the other direction, the only possible transition for $[SKIP]$ is $[SKIP] \xrightarrow{\tau} SKIP$ according to rules: (prefix) and (hid2) in Figure 2.2.

This transition is matched by: $SKIP \Longrightarrow SKIP$.

Therefore, according to Definition 2.3 $SKIP \approx [SKIP]$.

2. $p = a?{>}x \rightarrow p$, $[p] = (iPoF(a)?x \rightarrow [\![p]\!]) \setminus \{term\}$

   The only possible transition for $(a?{>}x \rightarrow p)$ is: $(a?{>}x \rightarrow p) \xrightarrow{a\leftarrow.v} p[v/x]$ according to rule (asy-in) in Section 3.2.

   This transition is matched by the encoded process in the following way:

   $[a?{>}x \rightarrow p] \xrightarrow{\tau}\xrightarrow{a\leftarrow.v} [p][v/x]$ according to (nFuns) in Section 2.4.3, then according to rules: (prefix-in) in Section 3.1 and (hid1) in Figure 2.2.

   In the other direction, the only possible transition for $[a?{>}x \rightarrow p]$ is as follows:

   We know that, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ by Lemma 3.1.

   This implies that, if $[p] \xrightarrow{\tau} [p']$ then $(iPoF(a)?x \rightarrow [\![p]\!]) \setminus \{term\} \xrightarrow{\tau} (a{\leftarrow}.x \rightarrow [\![p]\!]) \setminus \{term\}$ according to (nFuns) in Section 2.4.3.

   This transition is matched by the CSPa process in the following way:

   $(a?{>}x \rightarrow p) \Rightarrow (a?{>}x \rightarrow p)$.

   Therefore, according to Definition 2.3 $(a?{>}x \rightarrow p) \approx [a?{>}x \rightarrow p]$

3. $p = a!{<}x \rightarrow p$, $[p] = (oPiF(a)!x \rightarrow [\![p]\!]) \setminus \{term\}$

   The only possible transition for $(a!{<}x \rightarrow p)$ is: $(a!{<}x \rightarrow p) \xrightarrow{a\rightarrow.x} p$ according to rule (asy-out) in Section 3.2.

   This transition is matched by the encoded process in the following way:

   $[a!{<}x \rightarrow p] \xrightarrow{\tau}\xrightarrow{a\rightarrow.x} [p]$ according to (nFuns) in Section 2.4.3, then according to rules: (prefix-out) in Section 3.1 and (hid1) in Figure 2.2.

   In the other direction, the only possible transition for $[a!{<}x \rightarrow p]$ is:

   We know that, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ by Lemma 3.1.

This implies that, if $[p] \xrightarrow{\tau} [p']$ then $(oPiF(a)!x \rightarrow [\![p]\!]) \setminus \{term\} \xrightarrow{\tau} (a{\rightarrow}.x \rightarrow [\![p]\!]) \setminus \{term\}$ according to (nFuns) in Section 2.4.3.

This transition is matched by the CSPa process in the following way:

$(a!{<}x \rightarrow p) \Rightarrow (a!{<}x \rightarrow p)$.

Therefore, according to Definition 2.3 $(a?{>}x \rightarrow p) \approx [a?{>}x \rightarrow p]$

4. $f = p\|_A q$, $[f] = ([\![p]\!] \parallel_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$

   The possible transitions for $(p\|_A q)$ are as follows:

   - If $p \xrightarrow{\sigma} p'$ then $(p\|_A q) \xrightarrow{\sigma} (p'\|_A q)$ by rule (par1) in Figure 2.2; where $\sigma \notin A$ and $\sigma \neq term$.

     This transition is matched by the encoded process in the following way:

     (a) If $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then
     We know that, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ by Lemma 3.1.
     This implies that, if $[\![p]\!] \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} [\![p']\!]$ then $([\![p]\!] \|_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$ $\xrightarrow{\tau}{}^{*} \xrightarrow{\sigma}$ $([\![p']\!] \|_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$ according to (nFuns) in Section 2.4.3, and then by rules: (par1) and (hid1) in Figure 2.2.

     (b) if $\sigma \in \Sigma$ then:
     We know that, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ by Lemma 3.1.
     This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ then $([\![p]\!] \|_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$ $\xrightarrow{\sigma}$ $([\![p']\!] \|_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$ by rules: (par1) and (hid1) in Figure 2.2.

   - If $q \xrightarrow{\sigma} q'$ and then $(p\|_A q) \xrightarrow{\sigma} (p\|_A q')$ by rule (par2) in Figure 2.2; where $\sigma \notin A$ and $\sigma \neq term$.

     This transition is matched by the encoded process in the following way:

     (a) If $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then
     We know that, if $[q] \xrightarrow{\sigma} [q']$ then $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 3.1.
     This implies that, if $[\![q]\!] \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\}\cup A} [\![q]\!]) \setminus \{term\}$ $\xrightarrow{\tau}{}^{*} \xrightarrow{\sigma}$ $([\![p]\!] \|_{\{term\}\cup A} [\![q']\!]) \setminus \{term\}$ according to (nFuns) in Section 2.4.3, and then by rules: (par2) and (hid1) in Figure 2.2.

     (b) if $\sigma \in \Sigma$ then:

We know that, if $[q] \xrightarrow{\sigma} [q']$ then $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 3.1.
This implies that, if $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\}$
$\xrightarrow{\sigma} ([\![p]\!] \|_{\{term\} \cup A} [\![q']\!]) \setminus \{term\}$ by rules: (par2) and (hid1) in Figure 2.2.

- If $p \xrightarrow{\sigma} p'$ and $q \xrightarrow{\sigma} q'$ then $(p \|_A q) \xrightarrow{\sigma} (p' \|_A q')$ by rule (par3) in Figure 2.2; where $\sigma \in A$ and $\sigma \neq term$.

  This transition is matched by the encoded process in the following way:

  (a) If $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then
  We know that, if $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 3.1.
  This implies that, if $[\![p]\!] \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!])$
  $\setminus \{term\} \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} ([\![p']\!] \|_{\{term\} \cup A} [\![q']\!]) \setminus \{term\}$ if $\sigma \in A$ according to (nFuns) in Section 2.4.3, and then by rules (par3) and (hid1) in Figure 2.2.

  (b) if $\sigma \in \Sigma$ then:
  We know that, if $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 3.1.
  This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!])$
  $\setminus \{term\} \xrightarrow{\sigma} ([\![p']\!] \|_{\{term\} \cup A} [\![q']\!]) \setminus \{term\}$ by rules (par3) and (hid1) in Figure 2.2.

- If $p \xrightarrow{\surd} STOP$ and $q \xrightarrow{\surd} STOP$ then $(p \|_A q) \xrightarrow{\surd} STOP$ by rule (parST) in Section 3.2.

  This transition is matched by the encoded process in the following way:
  We know that, if $[p] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ and $[q] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ then $[\![p]\!] \xrightarrow{\tau}{}^{*}$
  $\xrightarrow{term} \xrightarrow{\surd} STOP$ and $[\![q]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ by Lemma 3.1.
  This implies that, if $[\![p]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ and $[\![q]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\} \xrightarrow{\tau}{}^{*} STOP$ by rules: (par3), (hid2), (parT1), (hid1), (parT2), (hid1), (parT3), and (hid3) in Figure 2.2.

In the other direction, since $[p \|_A q] = ([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\}$ by Definition

3.5, the possible transitions for $[p \|_A q]$ are:

- If $\sigma \notin \{term\}$ and $[p] \xrightarrow{\sigma} [p']$, then by Lemma 3.1, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$.

  This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p']\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\}$ where $\sigma \notin A$ by rules: (par1) and (hid1) in Figure 2.2.

  This transition is matched by the CSPa process in the following way:

  If $\sigma \in \Sigma$, or $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then, if $p \xrightarrow{\sigma} p'$ then $(p \|_A q) \xrightarrow{\sigma} (p' \|_A q)$ if $\sigma \notin A$ by rule (par1) in Figure 2.2.

  Alternatively, if $\sigma = \tau$ then $(p \|_A q) \Rightarrow (p \|_A q)$ if $\tau$ comes from executing functions; otherwise $(p \|_A q) \xrightarrow{\tau} (p' \|_A q)$

- If $\sigma \notin \{term\}$ and $[q] \xrightarrow{\sigma} [q']$ then by Lemma 3.1, if $[q] \xrightarrow{\sigma} [q']$ then $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$.

  This implies that, if $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p]\!] \|_{\{term\} \cup A} [\![q']\!]) \setminus \{term\}$ if $\sigma \notin A$ by rules: (par2) and (hid1) in Figure 2.2.

  This transition is matched by the CSPa process in the following way:

  If $\sigma \in \Sigma$, or $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then, if $q \xrightarrow{\sigma} q'$ then $(p \|_A q) \xrightarrow{\sigma} (p \|_A q')$ if $\sigma \notin A$ by rule (par2) in Figure 2.2.

  Alternatively, if $\sigma = \tau$ then $(p \|_A q) \Rightarrow (p \|_A q)$ if $\tau$ comes from executing functions; otherwise $(p \|_A q) \xrightarrow{\tau} (p' \|_A q)$

- If $\sigma \notin \{term\}$, $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then by Lemma 3.1, if $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$

  This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p']\!] \|_{\{term\} \cup A} [\![q']\!]) \setminus \{term\}$ if $\sigma \in A$ by rules: (par3) and (hid1) in Figure 2.2.

  This transition is matched by the CSPa process in the following way:

  If $\sigma \in \Sigma$, or $\sigma \in \{a \leftarrow, a \rightarrow | a \in \Sigma\}$ then, if $p \xrightarrow{\sigma} p'$ and $q \xrightarrow{\sigma} q'$ then $(p \|_A q) \xrightarrow{\sigma} (p' \|_A q')$ if $\sigma \in A$ by rule (par3) in Figure 2.2.

  Alternatively, if $\sigma = \tau$ then $(p \|_A q) \Rightarrow (p \|_A q)$ if $\tau$ comes from executing functions; otherwise $(p \|_A q) \xrightarrow{\tau} (p' \|_A q)$

- If $\sigma = \{term\}$, $[p] \xrightarrow{\tau} SKIP$, and $[q] \xrightarrow{\tau} SKIP$ then by Lemma 3.1, if $[p] \xrightarrow{\tau} SKIP$ and $[q] \xrightarrow{\tau} SKIP$ then $[\![p]\!] \xrightarrow{term} SKIP$ and $[\![q]\!] \xrightarrow{term} SKIP$ This implies that, if $[\![p]\!] \xrightarrow{term} SKIP$ and $[\![q]\!] \xrightarrow{term} SKIP$ then $([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \setminus \{term\} \xrightarrow{\tau} (SKIP \|_{\{term\} \cup A} SKIP) \setminus \{term\}$ by rules: (par3) and (hid2) in Figure 2.2.

  This transition is matched by the CSPa process in the following way: $(SKIP \|_A SKIP) \Rightarrow (SKIP \|_A SKIP)$.

  Therefore, according to Definition 2.3 $(p \|_A q) \approx [p \|_A q]$

□

## 3.4 Conclusions and Related work

Asynchronous communications are crucial in environments with unreliable media, like the internet. Additionally, synchronous communications are important for the transfer of critical data. For this reason, in this chapter we define a calculus which supports mixed asynchronous and synchronous communications, by introducing an implicit buffer with each channel. We formally extended the syntax and the operational semantics of CSP with buffer loading and consuming primitives.

Buffers are a well-known mechanism for the implementation of asynchronous communications, and they have been extensively used in process calculi literature.

In CSP, the use of buffered channels to facilitate asynchronous communications has been previously discussed by Hoare [84]. This model has not been formally implemented, and assumes asynchronous communications only.

Buffered channels within CSP have been discussed also in [124], where all or a set of channels can be selected to be buffered between two processes. Installing buffers in two-way communications was enough for the purpose of this model in order to prove that the correctness of a network of processes is independent of the amount of buffering added. However, in our model, we aim to provide a practical model where asynchronous communications are available as primitive communications in addition to the standard synchronous and interleaving communications. Therefore, we extend CSP with built-in buffers which can be used anywhere in the system. Additionally, our model retains the multi-way communication mode of CSP.

CSP# (the input language of PAT [131] model checker) also extends the syntax of CSP with buffered channels to support mixed synchronous/asynchronous communications. However, CSP# does not support the generalised parallel composition operator, therefore all shared channels are communicating in synchronous mode with no option to change this. In our model, designers have the option to choose which channel to synchronise on.

An asynchronous version of an early CSP-based language was mentioned in [37]. However, our approach is different. Our calculus preserves the notion of an output guard, allowing the same process to use synchronous and asynchronous communications, whereas the language in [37] does not include an output guard, like in the asynchronous $\pi$-calculus [86], thus synchronous communications are not allowed. Moreover, the language in [37] has no generalised parallel composition and the parallel composition is only allowed at the top-level (see [113] for more details).

Early proposals which support asynchronous communications by forcing interactions between two processes to be always mediated by buffers are described in [60, 29]. More recent proposal is suggested in [27], where buffers have been introduced to the $\pi$-calculus [109] to facilitate asynchronous communications as an alternative to the no-output-guards approach implemented in [86]. In [27], the encodability between the two calculi has been studied. While we use the same concept by introducing buffers in the middle of communications, in our model, we buffered the communications between multiple processes and our calculus supports mixed synchronous/asynchronous communications.

In the context of SOC, to the best of our knowledge our model is the first to mix synchronous/asynchronous communications; other calculi support only synchronous communications [35, 36] or only asynchronous communications [100, 137].

# 4

# Mobile CSP ($\mathcal{M}$CSP)

## 4.1 Introduction

One of the significant features of complex systems is their ability to reconfigure themselves dynamically. Therefore, any process calculus proposed for modelling complex systems, specially SOC systems, should be equipped with primitives facilitating dynamic reconfiguration; more specifically, primitives to model delegation of the communication capabilities from one process to another. Passing channel names is a way of transferring communication capabilities, usually called *mobility*.

The most well-known theory to reason about this type of mobility is the $\pi$-calculus [109], the mobile extension of CCS [107]. Mobility is introduced into CCS by enabling the communication of channel names: if channel names are sent through other channels then these communication channels are delegated to the receiver. In the $\pi$-calculus, communicating channel names are governed by a set of rules to ensure names freshness and to avoid collisions between names in processes' alphabets, which grow silently in the calculus.

The mobility model of the $\pi$-calculus provides a simple and expressive theory to reason about systems where the dominant communications are *two-way communications* [124] (only two entities communicate), as in the *server-client architecture* [69]. This is because, in the $\pi$-calculus, only two complementary processes synchronise in the parallel composition.

When the dominant communications in a system are *multi-way* [124] (multiple entities communicate), as in the *service-oriented architecture* [67], then new notions

have to be introduced to the $\pi$-calculus, such as sessions [53, 35], or sites [45], to model these communications. On the other hand, calculi like CSP, which admit *multi-way communications*, lack mobility, which hinders their use in applications where entities' linkage networks are subject to change. Therefore, in this chapter, we propose a new model to accommodate mobility into CSP, namely Mobile CSP ($\mathcal{M}$CSP).

Our proposal extends the channel-semantics of CSP to allow channels to carry channel names, and includes mechanisms to change the interface set of parallel compositions as the participants communicate. To deal with the latter, we introduce a novel dynamic algorithm to update the interface set of the parallel composition (the silent growth of alphabets as in the $\pi$-calculus is not enough). By using our algorithm, in $\mathcal{M}$CSP, processes participating in the parallel composition not only can choose between synchronous and interleaving communications, as in CSP, but they can dynamically switch between the two modes.

Formally speaking, we extend the CSP channel syntax to include a definition of mobile channels, which facilitates the updating of processes' base knowledge, and we provide an operational semantics which explains how the parallel composition interface set is updated and how channels are maintained. Additionally, we study the relationship between our new calculus and the $\pi$-calculus by encoding the $\pi$-calculus into $\mathcal{M}$CSP. We prove that there is an operational correspondence [77] between the two theories.

### Contributions in this chapter

1. We formally introduce mobile channel communications into CSP, by permitting channels to carry other channels.

2. In addition to the usual notion of mobility where channels carry other channels, our model provides another notion of mobility whereby the communication mode between channels can be switched from synchronous to interleaving and via versa.

3. We develop a full channel operational semantics for CSP where data can be integer or channel.

4. We study the relationship between $\mathcal{M}$CSP and the $\pi$-calculus, and we prove that the two theories are operationally correspondent.

**Structure of this chapter** Section 2.4.4 recalls the $\pi$-calculus. Section 4.2 introduces our mobility model for CSP ($\mathcal{M}$CSP). Section 4.3 presents the syntax and the operational semantics of $\mathcal{M}$CSP. Section 4.4 investigates the relation between $\mathcal{M}$CSP and the $\pi$-calculus. Finally, Section 4.5 concludes the chapter and discusses related works.

## 4.2 A Mobility Model for CSP

To achieve mobility while maintaining the expressivity of the calculus, we will allow channel names to be passed as part of the data sent on channels. We identify communicated channel names by enclosing them in *brackets*. For instance, $a?x!5?(c)$ denotes a communication through the channel $a$ where the process accepts input into variable $x$, outputs the value 5, and finally accepts the new connection $c$. There is no restriction on the number of components sent through channels. For instance, $a!(c)?(m)$ is a valid event in $\mathcal{M}$CSP.

Formally speaking, we extend the standard channel syntax of CSP with the following types of events: (i) *Output mobile channel* is written as $a!(c)$, where $a$ is the current channel and $c$ is the communicated channel name. (ii) *Input mobile channel* is written as $a?(m)$, where $a$ is the current channel and $m$ is a mobile channel name (variable).

One of the main differences between the $\pi$-calculus and CSP is that in CSP the parallel composition is parameterised with an interface set, which should be explicitly updated to reflect changes in the system, if mobility is permitted.

To facilitate updating the parallel composition's interface set (i.e. $A$ in $p \parallel_A q$), we optionally decorate mobile channel events with marks $+$ and $-$. These marks are used to guide changes to the interface set as follows:

- If events of type $a\_(c)^+$ are used, then the interface set $A$ will be augmented by the new communication (i.e. $A$ becomes $A \cup \{c\}$). As a result, participants will synchronise on the new communicated channel (i.e. $c$).

- If events of type $a\_(c)^-$ are used, then the communicated name ($c$) should be removed from the interface set $A$ (i.e. $A$ becomes $A - \{c\}$). As a result, participants will evaluate the new channel independently (interleaving).

- If *"no change"* events $a\_(c)$ are used, then the interface set $A$ remains the same.

Here, $a\_(c)$ denotes the set of events: $a!(c)$, $a?(m)$, and $a.(c)$.

In order for these effects to take place the carrying channel should be part of the interface set. More specifically, if the carrying channel is part of the interface set, then the mobile channel will be delivered to the recipients and the interface set will be updated. On the other hand, if the carrying channel is not part of the interface set, then the mobile channel will not be delivered to the recipients and the interface set will not be updated. This is consistent with CSP's original communication model, where channels in the interface set synchronise and their data is delivered, and channels not in the interface set interleave and their data is lost. In other words, if a channel is part of the interface set then all events containing this channel name will be performed in a synchronous mode, so if it carries another channel name then this name will be delivered to the recipients and the effects of mobile channel synchronisation on the interface set should be as follows:

- If $a\_(c)^+$ synchronises with $a\_(c)^+$, then the interface set will be updated with the new name ($A := A \cup \{c\}$).

- If $a\_(c)^+$ synchronises with $a\_(c)$, then the interface set will be updated with the new name ($A := A \cup \{c\}$).

- If $a\_(c)^-$ synchronises with $a\_(c)^-$, then the communicated name will be removed from the interface set ($A := A - \{c\}$).

- If $a\_(c)^-$ synchronises with $a\_(c)$, then the communicated name will be removed from the interface set ($A := A - \{c\}$).

- If $a\_(c)$ synchronises with $a\_(c)$, then the interface set remains the same.

- The synchronisation between $a\_(c)^+$ and $a\_(c)^-$ is prohibited in our model and whenever encountered, processes will be blocked.

In the last case, we prefer to block processes rather than enforcing either of the communication modes, because enforcing a certain mode of communication might conflict with the process's capabilities and requirements. Instead of blocking processes, a warning message could be sent back (then the processes can decide). However, this solution complicates the calculus, and we propose to investigate it in future work.

*Remark.* Decorating channels with $(+, -)$ markings is inspired by [124]. However, the marking in our model affects the synchronisation set whereas in [124] affects the process alphabets only, which results in two different models. More precisely, in [124], an alphabetised version of the parallel composition is used where a channel carrying/accepting a mobile channel with $(+)$ marking synchronises either with the same channel carrying/accepting a mobile channel with $(+)$ marking, or with the same channel carrying/accepting a mobile channel with $(-)$ marking. In the first case, both should have the mobile channel in their alphabets so both will always synchronise on the new channel. In the second case, the one marked with $(+)$ has the mobile channel in its alphabet and the other does not, so the one having the mobile channel can communicate independently on it, and the one without the mobile channel cannot communicate on this channel any longer. As a consequence, in the model defined in [124], we cannot express a situation where both processes have the mobile channel and use it to communicate on this channel independently, as Example 4.2 illustrates. Similarly, the case where both processes in the parallel composition insist to not synchronise on the new channel or one insists and the other does not care cannot be directly expressed in the model proposed in [124].

An important consideration when channels are passed is whether or not they modify the process alphabet. In our model, inputting processes' alphabets will be updated with the new channel name if it is not already there. Outputting processes' alphabets will not be affected. However, if a process wants to send a connection and release it (i.e. removing it from its alphabet), then output channels $(a!(c)^-)$ should be tagged with a star $a!(c)^{-*}$. Although processes' alphabets do not affect the interface set in the generalised parallel composition, we include this option so our mobility model can work with CSP's alphabetised parallel composition (i.e. $p \,{}_{\alpha p}\|_{\alpha q}\, q$) as well.

The following examples illustrate the usability of our model.

**Example 4.1.** Our model implements the usual notion of mobility, by sending channel names through existing names. The $\pi$-calculus process $\bar{a}b.\mathbf{0}|a(x).\bar{x}c.\mathbf{0}$ can be written in $\mathcal{M}$CSP as $(a!(b)^+ \rightarrow STOP) \parallel_{\{|a|\}} (a?(x)^+ \rightarrow x!(c)^+ \rightarrow STOP$, where *brackets* are used to identify channel names, and the $(+)$ marks are used because in the $\pi$-calculus communications are always synchronous (hence, we should add the communicated channel to the interface set).

**Example 4.2.** Our model can also implement other notions of mobility, which do not necessarily change the linkage network of a process, but change the mode of communication over these names. Consider the process: $((a!(b)^+ \rightarrow b!5 \rightarrow a!(b)^- \rightarrow b!3 \rightarrow SKIP) \parallel_{\{|a|\}} (a?(x)^+ \rightarrow x?y \rightarrow a?(x)^- \rightarrow x?z \rightarrow SKIP)$. The first emission of name $b$ adds $b$ to the interface set of the parallel composition, so 5 is sent synchronously. The second emission of name $b$ removes $b$ from the interface set of the parallel composition, therefore, $b!3$ and $b?z$ are evaluated independently, Here, $x?z$ becomes $b?z$ because the last instantiation of the variable $x$ in this process was $b$.

**Example 4.3.** The *"no change"* events are useful in cases when the communication mode is determined by one process. Let $C_1 = com?(sup)^+ \rightarrow sup!order \rightarrow release!(sup)^{-*} \rightarrow SKIP$ be a customer process which is ordering items from a supplier. Let $S_1 = com!(freeCh) \rightarrow freeCh?orderC \rightarrow release!(freeCh)^- \rightarrow SKIP$ be the supplier process. Let the two processes run in parallel $C_1 \parallel_{\{|com|\}} S_1$. In this example, the supplier sends a free connection to the customer (channel $freeCh$) and lets it decide the communication mode (according to its capabilities and requirements), then it accepts the order and releases the connection. The customer accepts the channel (input into variable *sup*) and after sending the order, it releases the connection and forgets it completely (i.e. this connections will be removed from the process's alphabet).

## 4.3 $\mathcal{M}$CSP Semantics

In this section we develop the formal semantics of $\mathcal{M}$CSP. We first present the syntax of $\mathcal{M}$CSP, then its operational semantics.

### 4.3.1 $\mathcal{M}$CSP syntax

The syntax of $\mathcal{M}$CSP is given in Figure 4.1. $\mathcal{M}$CSP extends CSP with primitives to facilitate mobile channel communications.

Events in $\mathcal{M}$CSP, as in the standard CSP, can be classified as *simple events*, which are single literal names; *compound events*, which are literal names combined with other literal names or variables via the operator (.), e.g., $a$ or $a.b$; or *communication events*, which are simple or compound events composed with integer expressions to denote communicated data. Although the standard CSP allows data of different types to be communicated through channels, in soaCSP semantics, for simplicity, we assume that the communicated data are integers and literal names (channel names).

The novelty in $\mathcal{M}$CSP is that channels can carry other channels. This is achieved by enclosing channel names within *brackets* and optionally decorating them with $(+, -, *)$ marks. The marks are used to allow processes to notify the parallel composition of the mode of communication on the communicated name.

Additionally, we extend the parallel composition syntax by allowing the indexed version of it to be indexed with infinity. This is necessary to facilitate an accurate encoding from $\pi$ to $\mathcal{M}$CSP, as shown in Section 4.4.

### 4.3.2 $\mathcal{M}$CSP Operational Semantics

The operational semantics of $\mathcal{M}$CSP extends the operational semantics of CSP. We only present the semantics of the new mobility extensions since the rest of $\mathcal{M}$CSP is similar to CSP (see Chapter 2).

We start by presenting a detailed semantics for channels. For the sake of simplicity, we assume that channels can only carry integers or channel names; extensions to other types of data are left for future work. Channel names are arbitrary literals; below we use $c$ to denote constant channel names and $m$ to denote variable channel names.

In the standard CSP [124], channels carrying data represent a class of events in $\Sigma$ rather than one single value, e.g., $a.\mathbb{Z}$ represents the channel $a$ which can communicate any integer value ($a.\mathbb{Z}$ means $\{a.x | x \in \mathbb{Z}\} \subseteq \Sigma$). Inputting processes are able to receive any integer value, whereas outputting processes are only able to communicate one value. Therefore, if $a.\mathbb{Z}$ is an input event then it is actually an external choice out of all the values of type integer, which is represented succinctly as $a?x$. If $c.\mathbb{Z}$

(Standard Processes)
$p, q ::= \ldots$ (CSP syntax, see Figure 2.2) $\mid \|_{A}^{N}{}_{i=1}\ p_i$ where $N \in \mathbb{N} \cup \{\infty\}$

(events)
$a ::=$ $name?INPC$ (input) $\mid$ $name!OUTC$ (output)
$\mid$ $name.OUTC$ (compound) $\mid$ $name$ (literal name)

(Input components)
$INPC ::=$ $INP \mid INP?INPC \mid INP!OUTC \mid INP.OUTC$
(Input elements)
$INP ::=$ $name$ (literal name ) $\mid$ $\ell$ (integer variable)
$\mid$ $(name)$ (neutral com) $\mid$ $(name)^{+}$ (synchronous com)
$\mid$ $(name)^{-}$ (interleaving com)

(Output components)
$OUTC ::=$ $OUT \mid OUT?INPC \mid OUT!OUTC \mid OUT.OUTC$
(Output elements)
$OUT ::=$ $name$ (literal name ) $\mid$ $e$ (integer expression)
$\mid$ $(name)$ (neutral com) $\mid$ $(name)^{+}$ (synchronous com)
$\mid$ $(name)^{-}$ (interleaving com)
$\mid$ $(name)^{-*}$ (send channel then remove it from $\alpha p$ )

Figure 4.1: $\mathcal{M}$CSP Syntax

is an output event then it is a single integer value, $a.x$ where $x \in \mathbb{Z}$. Output events are represented as $a!x$. In our semantics, $a.x$ always equals $a!x$, except when the $x$'s value is not known then this event is a declaration of a new input variable.

Similarly, in $\mathcal{M}$CSP, channels carrying other channels represent a class of events in $\Sigma$.

**Definition 4.1.** *$a.CH$* denotes a channel $a$ that can communicate any channel name in $\Sigma$, that is

$$a.CH = \{a.(b)^{\varphi} | b \in channels(\Sigma) \wedge \varphi \in \{\emptyset, +, -, -*\}\}$$

where $channels(\Sigma)$ is a generalisation of the $channel(a.v)$ function [84], which extracts the channel name of an event, that is, $channels(\Sigma)$ will return the set of all channel names of $\Sigma$'s events. If $\varphi = \emptyset$, the mark is omitted (e.g., $a.(c)$).

Throughout what follows, if $a$ is a channel in $\Sigma$ then $\{|a|\}$ is the enumeration of

$a$'s class of events in $\Sigma$.

Inputting processes are able to receive any channel name in $\Sigma$, whereas outputting processes are only able to communicate one channel name.

In $\mathcal{M}$CSP, the LTS is defined as a tuple $(((MCSP, \sigma), \mathcal{P}^{\Sigma}), LTSlabels, \longrightarrow)$, where $MCSP$ denotes the full space of $\mathcal{M}$CSP processes, $\mathcal{P}^{\Sigma}$ is the set of parts of $\Sigma$, LTSlabels denotes the set of transition labels, and $\longrightarrow \subseteq ((MCSP, \sigma), \mathcal{P}^{\Sigma}) \times LTSlabels \times ((MCSP, \sigma), \mathcal{P}^{\Sigma})$. If $p, q \in MCSP$, $\alpha p, \alpha q \in \mathcal{P}^{\Sigma}$, and $a \in LTSlabels$, then $(((p, \sigma), \alpha p), a, ((p, \sigma), \alpha q)) \in \longrightarrow$ can be written as $((p, \sigma), \alpha p) \xrightarrow{a} ((q, \sigma), \alpha q)$. For the sake of clarity, we omit the second component of the LTS configurations, writing transitions simply as $(p, \sigma) \xrightarrow{a} (q, \sigma)$; we explicitly indicate with $(*)$ the rules that change the second component.

As can be seen in the LTS of $\mathcal{M}$CSP, the basic configuration (denoted by $C$) is a pair $(p, \sigma)$, where $\sigma$ is a local store. $\sigma$ is a collection of integer locations and literal name locations. We use $\sigma$, $\sigma'$,... to represent its different states. We write $\sigma(x)$ to denote the value of the variable $x$ in $\sigma$.

However, this form of configuration is not yet sufficient. Consider the case where a process $p$ is composed in parallel with $q$ (i.e. $(p \|_A q, \sigma)$). Assume $p$ performs an event $x$, then there is a transition from this initial configuration to a new configuration reflecting this change. Following the (par1) rule in Chapter 2 (updated with the store):

$$\frac{(p, \sigma) \xrightarrow{a} (p', \sigma')}{(p \ \|_A \ q, \sigma) \xrightarrow{a} NC}$$

where NC is a new configuration which has the process $p'$ in $\sigma'$ composed in parallel with $q$ which is still in $\sigma$. Since $\sigma'$ and $\sigma$ may differ, the composition operators (which were defined to work on processes) will be promoted to work on configurations. We use $\psi$ to denote the full space of configurations. The promoted versions of operators (where processes' stores might differ) are defined as follows:

$$\psi, \psi' \ ::= \ C \mid \psi \Box \psi' \mid \psi \|_A \psi' \mid \psi; \psi' \mid \psi \backslash a \mid \psi [\![ R ]\!] \mid \psi ||| \psi$$

Note that, the state of the local store of a process changes if mobile channels or data integers are communicated. Apart from this the store remains the same. In the semantics, $e$ is evaluated according to the standard integer semantics. Literal names

can be any name, however for the sake of clarity *we use c to denote constant channel names, and we use m to denote variable mobile channel names.*

According to CSP's syntax (see Chapter 2), events in $\Sigma$ can be evaluated within the prefix operator as rule (prefix) in Figure 2.2 shows. However, this is not sufficient if $a$ is a channel passing variables, either integer or channel names. Therefore, below we update the definition of the *prefix operator* to implement the other types of events as follows:

- The (out1,2),(out3), (ud.out1,2) and (ud.out3) rules below show that if the event is an output channel $a$ that carries data $e$ (i.e. $a!e$ or $a.e$) then $e$ is evaluated and afterwards its value is sent.

$$\text{(out1,2)}\ \frac{(e,\sigma) \xrightarrow{\tau} (e',\sigma)}{(a!e \to p,\sigma) \xrightarrow{\tau} (a!e' \to p,\sigma)}\ \frac{}{(a!\ell \to p,\sigma) \xrightarrow{\tau} (a!\sigma(\ell) \to p,\sigma)}$$

$$\text{(out3)}\frac{}{(a!n \to p,\sigma) \xrightarrow{a.n} (p,\sigma)}\ n \in Z$$

$$\text{(ud.out1,2)}\frac{(e,\sigma) \xrightarrow{\tau} (e',\sigma)}{(a.e \to p,\sigma) \xrightarrow{\tau} (a.e' \to p,\sigma)}\ \frac{}{(a.\ell \to p,\sigma) \xrightarrow{\tau} (a.\sigma(\ell) \to p,\sigma)}\ \ell \in \sigma$$

$$\text{(ud.out3)}\frac{}{(a.n \to p,\sigma) \xrightarrow{a.n} (p,\sigma)}\ n \in Z$$

- The (in) and (ud.in) rules below show that if the event is an input channel $a$ that expects data $\ell$ (i.e. $a?\ell$ or $(a.\ell\ \wedge\ \ell \notin \sigma)$), then the store will be updated with the received value.

$$\text{(in)}\ \frac{}{(a?\ell \to p,\sigma) \xrightarrow{a.n} (p,\sigma[\ell \mapsto n])}\ n \in Z$$

$$\text{(ud.in)}\frac{}{(a.\ell \to p,\sigma) \xrightarrow{a.n} (p,\sigma[\ell \mapsto n])}\ \ell \notin \sigma \wedge n \in Z$$

- (ch-out1,2 , ch-ud.out1,2) rules below show that if the event is an output channel $a$ that carries variable channel name $m$ (i.e. $a!(m)$, $a!(m)^-$, $a!(m)^+$, $a!(m)^{-*}$, $a.(m)$, $a.(m)^-$, $a.(m)^+$, or $a.(m)^{-*}$), then its value $c$ will be retrieved from the

store $\sigma$ before sending it.

$$\text{(ch-out1)}\frac{}{(a!(m)^{\varphi} \to p, \sigma) \xrightarrow{\tau} (a!(\sigma(m))^{\varphi} \to p, \sigma)}$$

$$\text{(ch-out2*)}\frac{}{(a!(c)^{\varphi} \to p, \sigma) \xrightarrow{a.(c)^{\varphi}} (p, \sigma)}\ c \in channels(\Sigma)$$

$$\text{(ch-ud.out1)}\frac{}{(a.(m)^{\varphi} \to p, \sigma) \xrightarrow{\tau} (a.(\sigma(m))^{\varphi} \to p, \sigma)}\ m \in \sigma$$

$$\text{(ch-ud.out2*)}\frac{}{(a.(c)^{\varphi} \to p, \sigma) \xrightarrow{a.(c)^{\varphi}} (p, \sigma)}\ c \in channels(\Sigma)$$

where $()^{\varphi}$ can be $()$, $()^{+}$, $()^{-}$, or $()^{-*}$.

If the $(-*)$ marking is used then *the alphabet of the process is updated by removing this channel from it*, e.g. if $p$ communicates $(c)^{-*}$ and evolves to $p'$ then the alphabet of $p'$ should not have $c$. More precisely, $(p, \alpha p) \xrightarrow{a.(c)^{-*}} (p', \alpha p')$ and $\alpha p' = \alpha p - \{c\}$.

- (ch-in, ch-ud.in) rules below show that if the event is an input channel $a$ that expects channel name $m$ (i.e. $a?(m)$, $a?(m)^{+}$, $a?(m)^{-}$, $a.(m)$, $a.(m)^{+}$, or $a.(m)^{-}$ if $m \notin \sigma$), then the store will be updated with the received value.

$$\text{(ch-in*)}\frac{}{(a?(m)^{\varphi} \to p, \sigma) \xrightarrow{a.(c)^{\varphi}} (p, \sigma[m \mapsto c])}\ c \in channels(\Sigma)$$

$$\text{(ch-ud.in*)}\frac{}{(a.(m)^{\varphi} \to p, \sigma) \xrightarrow{a.(c)^{\varphi}} (p, \sigma[m \mapsto c])}\ m \notin \sigma \wedge c \in channels(\Sigma)$$

where $()^{\varphi}$ can be $()$, $()^{+}$, or $()^{-}$.

Upon receiving the channel name, *the alphabet of the process is updated by adding this channel to it, if it is not already there*, e.g. if $p$ receives $c$ in $(m)^{\varphi}$ and evolves to $p'$, then if $c \notin \alpha p$ the alphabet of $p'$ should be updated with $c$. More precisely, $(p, \alpha p) \xrightarrow{a.(c)^{\varphi}} (p', \alpha p')$ and $\alpha p' = \alpha p \cup \{c\}$.

- (var-ch1,2,3) rules below show that if the event is a channel variable $m$ (i.e. $m?INPC$, $m!OUTC$, or $m.OUTC$), then its value will be retrieved from the

store $\sigma$ before evaluating it.

$$\text{(var-ch1)}\frac{}{(m?INPC \to p, \sigma) \xrightarrow{\tau} (\sigma(m)?INPC \to p, \sigma)}$$

$$\text{(var-ch2)}\frac{}{(m!OUTC \to p, \sigma) \xrightarrow{\tau} (\sigma(m)!OUTC \to p, \sigma)}$$

$$\text{(var-ch3)}\frac{}{(m.OUTC \to p, \sigma) \xrightarrow{\tau} (\sigma(m).OUTC \to p, \sigma)}$$

- the (prefix) rule below is the (prefix) rule in Figure 2.2 updated with stores.

$$\text{(prefix)}\frac{}{(a \to p, \sigma) \xrightarrow{a} (p, \sigma)}$$

For the sake of clarity, in the operational semantics we give rules for channels that carry only one data component. However, this can be extended to allow the generalised form of events, i.e., $a?INPC$, $a!OUTC$, or $a.OUTC$.

Processes can communicate and exchange information if they are composed in parallel. $\mathcal{M}$CSP, as the standard CSP, supports two modes of communications, *synchronous* and *interleaving*. If the communicated event is in the interface set of the parallel composition then the communication is carried synchronously, otherwise events will be evaluated independently (interleaving). Therefore, if data (or names) are intended to be transferred between processes then the carrying channel should be in the interface set otherwise the data (or names) will be lost. Synchronisation implements handshaken communication [124], where a communication can only happen if all its participants are prepared to execute this communication. In $\mathcal{M}$CSP, a communication is guided by an interface set $A$ which can be dynamically updated if mobile channels are communicated. $A$ is always assumed to be a sub-set of the events that are shared between the processes participating in the parallel composition, i.e. $A \subseteq \alpha p \cap \alpha q$, if $p$ and $q$ are processes composed in parallel. $\mathcal{M}$CSP follows CSP in assuming that $A \subseteq \alpha p \cap \alpha q$, if $p$ and $q$ are processes composed in parallel, without forcing it in the rules, so $A$ can have events not in $\alpha p \cap \alpha q$. However, having events in $A$ not in $\alpha p \cap \alpha q$ might cause synchronisation problems. For instance, if we have the following process $a \to b \to SKIP \parallel_{\{a,b\}} a \to SKIP$, then the process can evaluate $a$ as both sides of the parallel composition can perform $a$, but the process will block

on $b$ as only one side can perform $b$ and synchronisation is required on $b$. At the beginning, $A$ will be provided by the system designer.

In the following, we update the definition of CSP's parallel composition to implement the behaviour of this operator in case the new types of events (channels carrying variables) are communicated. First, the rules (par1,2) are similar to the rules (par1,2) in Figure 2.2 updated with stores, where $a$ in these rules can be of any event type. Similarly, the rules (parT1,2,3) in Figure 2.2 can be updated with stores.

$$\text{(par1,2)}\frac{(p,\sigma) \xrightarrow{a} (p',\sigma')}{(p \parallel_A q,\sigma) \xrightarrow{a} (p',\sigma') \parallel_A (q,\sigma)} \quad \frac{(q,\sigma) \xrightarrow{a} (q',\sigma')}{(p \parallel_A q,\sigma) \xrightarrow{a} (p,\sigma) \parallel_A (q',\sigma')}$$

where, $a \notin A$ and $a$ can be $b$, $b.n$, or $b.(c)^{\varphi}$; $\varphi$ as previously defined.

The rule (par3) is similar to the rule (par3) in Figure 2.2 updated with stores, where $a$ in this rule can be a simple event, a compound event, or a channel carrying data.

$$\text{(par3)}\frac{(p,\sigma) \xrightarrow{a} (p',\sigma') \quad (q,\sigma) \xrightarrow{a} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a} (p',\sigma') \parallel_A (q',\sigma'')}$$

where $a \in A$ and $a$ can be $b$ or $b.n$.

The rules (m-par1*) to (m-par6*) explain the behaviour of the parallel composition in case the communicated channel is carrying another channel name, and show how the interface set will be updated. In (m-par1*) to (m-par6*) rules, *if the $(-*)$ marking is communicated by $p$ or $q$ then their alphabets and the alphabet of the parallel composition are updated by removing this channel from them.* Moreover, *if a new channel is communicated by $p$ or $q$, then their alphabets will be updated and hence also the alphabet of the parallel composition is updated.* Note that no rule is provided for the case: $(a.(c)^{-}$ or $a!(c)^{-*}$ synchronise with $a.(c)^{+})$ because it is not allowed.

$$\text{(m-par1*)}\frac{(p,\sigma) \xrightarrow{a.(c)} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)} (p',\sigma') \parallel_A (q',\sigma'')}\ a \in A$$

$$\text{(m-par2*)}\frac{(p,\sigma) \xrightarrow{a.(c)^{+}} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)^{\varphi}} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)^{+}} (p',\sigma') \parallel_{A'} (q',\sigma'')}$$

where $a \in A$ , $A' = A \cup \{c\}$, and $()^{\varphi}$ can be $()$ or $()^{+}$.

$$\text{(m-par3*)}\frac{(p,\sigma) \xrightarrow{a.(c)} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)^{+}} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)^{+}} (p',\sigma') \parallel_{A'} (q',\sigma'')}$$

where $a \in A$ , $A' = A \cup \{c\}$.

$$\text{(m-par4*)}\frac{(p,\sigma) \xrightarrow{a.(c)} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)^{\varphi}} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)^{-}} (p',\sigma') \parallel_{A'} (q',\sigma'')}$$

where $a \in A$, $A' = A - \{c\}$, and $()^{\varphi}$ can be $()^{-}$, or $()^{-*}$.

$$\text{(m-par5*)}\frac{(p,\sigma) \xrightarrow{a.(c)^{-}} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)^{\varphi}} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)^{-}} (p',\sigma') \parallel_{A'} (q',\sigma'')}$$

where $a \in A$, $A' = A - \{c\}$, and $()^{\varphi}$ can be $()$, $()^{-}$, or $()^{-*}$.

$$\text{(m-par6*)}\frac{(p,\sigma) \xrightarrow{a.(c)^{-*}} (p',\sigma') \quad (q,\sigma) \xrightarrow{a.(c)^{\varphi}} (q',\sigma'')}{(p \parallel_A q,\sigma) \xrightarrow{a.(c)^{-}} (p',\sigma') \parallel_{A'} (q',\sigma'')}$$

where $a \in A$, $A' = A - \{c\}$, and $()^{\varphi}$ can be $()$, $()^{-}$, or $()^{-*}$.

The (par-inf) rule below explains the behaviour of the indexed version of the parallel composition if it is indexed to $\infty$. The event $a$ in this rule can be of any event type.

$$\text{(par-inf)}\frac{(p,\sigma) \xrightarrow{a} (p',\sigma')}{(\underset{A}{\parallel_{i=1}^{\infty}} p,\sigma) \xrightarrow{a} (p' \parallel_A (\underset{A}{\parallel_{i=1}^{\infty}} p,\sigma)}$$

## 4.4 Relation between the $\pi$-calculus and the $\mathcal{M}$CSP calculus

In this section, we investigate the expressivity of our mobility model by comparing it to the standard $\pi$-calculus. We are aware that there are extensions of the $\pi$-calculus with functions and integers, but we encode the standard $\pi$-calculus into $\mathcal{M}$CSP because in this chapter we focus on the expressivity of $\mathcal{M}$CSP's mobility

model rather than its other capabilities, like data passing.

We discuss the expressivity of our calculus by, first, encoding the $\pi$-calculus into $\mathcal{M}$CSP, then proving that there is an operational correspondence. Operational correspondence [77] captures the correctness of an encoding by checking the preservation of execution steps as defined in the operational semantics of the source and the target languages. It has been argued convincingly in [27] that operational correspondence is a suitable method to test the expressivity of theories. We emphasise that our aim is to show that the *name passing* concept, which is borrowed from the $\pi$-calculus and is implemented differently in our calculus, works correctly.

The notion of *operational correspondence* has been formally defined in [111, 77] as a notion of correctness; in [77] it is considered as a criterion for "good" encoding. We recall the definition from [77].

**Definition 4.2** (Operational Correspondence)**.** A translation $[\![.]\!] : \mathcal{L}_1 \rightarrow \mathcal{L}_2$, where ($\mathcal{L}_1, \mathcal{L}_2$ are the source and the target languages respectively) is an operational correspondence if it is:
Complete: for all $S \Longmapsto_1 S'$, where $S$ is a source language term, $[\![S]\!] \Longmapsto_2 \asymp_2 [\![S']\!]$;
Sound: for all $[\![S]\!] \Longmapsto_2 T$, where $T$ is a target language term, there exists an $S'$ such that $S \Longmapsto_1 S'$ and $T \Longmapsto_2 \asymp_2 [\![S']\!]$, where $_2 \asymp_2$ is a weaker formulation of equivalence.

According to [77], $\asymp_2$ is a behavioural equivalence needed to describe the abstract behaviour of a process. It has been mentioned in [77] that the formulation with $=$ is too demanding and the weaker formulation (with $\asymp$) is needed, because encodings may leave some "junk" after having mimicked some source language reduction; the use of $\asymp$ is justified to get rid of potential irrelevant "junk". However, it has been argued in [115] that this operational correspondence criterion can be trivial if the target term relation has not been fixed. For instance, any encoding is operational corresponding w.r.t the universal relation on target terms. Also, the conditions are trivially satisfied if the transitions are trivial, e.g. mapping every process to the process which does nothing.

In this chapter we use this criterion without the help of the relation $\asymp$; see the statements of Theorems 4.1 and 4.2. This clarifies that our encoding is not trivial, since it preserves transition labels.

### 4.4.1 From $\pi$-calculus to $\mathcal{M}$CSP

Defining a faithful encoding between two different theories that rely on different foundations, like CSP, or more specifically $\mathcal{M}$CSP, and the $\pi$-calculus, is challenging. Both $\mathcal{M}$CSP and the $\pi$-calculus have input and output prefixes and a similar set of operators; however there are some major differences which should be treated carefully in the encoding to ensure that the translated process will work exactly like the original process. These differences can be summarised in the following points:

- Events in CSP are multiparty events. That is, the events can be divided into several components and these components can be simultaneously conveyed in both directions. In the $\pi$-calculus, events have only one component which is transmitted in one direction at a time.
- $\mathcal{M}$CSP, like CSP, does not admit bound names, as all the names in the calculus can be seen by all processes.
- In the $\pi$-calculus, only two complementary processes are required to synchronise in the parallel composition. On the other hand, $\mathcal{M}$CSP, like CSP, requires all the processes in the parallel composition to synchronise on the shared events (or a subset of them).
- In the $\pi$-calculus, processes participating in the parallel composition should synchronise on shared events, whereas in $\mathcal{M}$CSP, as in CSP, processes may or may not synchronise on shared events, depending on whether these events are in the interface set of the parallel composition or not.
- Communications in the $\pi$-calculus are always synchronous, whereas in $\mathcal{M}$CSP, processes can choose, or later switch, between synchronous and interleaving communication.

We will start by defining a mapping between events in the two calculi, and some auxiliary functions. The encoding of processes will be defined afterwards.

**Mapping names** As mentioned in Section 2.4.3 and Section 2.4.4, $n(p)$ denotes the set of names in $\pi$-calculus process $p$, and $\Sigma$ denotes the universal set of events in

$\mathcal{M}$CSP. In the $\pi$-calculus, only names mentioned in prefixes, e.g. $a$ and $b$ in $a(b)$, are in the set of process names. However, in $\mathcal{M}$CSP, as in CSP, the complete events, e.g. $a.(b)$, are included in the set $\Sigma$ and in process alphabets.

- [**Calculating the set** $\Sigma$] Considering the description of $\Sigma$ in Section 4.3.2 (see Definition 4.1), we define the initial set $\Sigma$ for the encoding of a $\pi$-calculus process $p$ with names $n(p)$ as:

$$\Sigma_0 = \{a.(b) \mid a \in n(p) \; and \; b \in n(p)\}$$

This set will be updated by using the function ***update***$\Sigma$, which accepts a name $a$ and updates the current $\Sigma_i$, returning $\Sigma_{i+1}$. It is defined as follows:

$$\begin{aligned} update\Sigma(a, \Sigma_i) = \Sigma_i \quad &\cup \quad \{a.(c) \mid c \in channels(\Sigma_i)\} \\ &\cup \quad \{c.(a) \mid c \in channels(\Sigma_i)\} = \Sigma_{i+1} \end{aligned}$$

We call $\Sigma$ the union of $\Sigma_0, \Sigma_1, \ldots$. Consider the following example for illustration:

**Example 4.4.** Let $f \stackrel{\text{def}}{=} p + q$, where $p \stackrel{\text{def}}{=} \bar{a}c.\tau.b(m).\mathbf{0}$ and $q \stackrel{\text{def}}{=} a(k).\bar{b}d.\mathbf{0}$

be a $\pi$-calculus process, then $n(f) = n(p) \cup n(q) = \{a, b, c, m, k, d\}$ where $n(p) = \{a, b, c, m\}$ and $n(q) = \{a, b, k, d\}$.

Thus, $\Sigma_0 = \{x.(y) | x \in n(f) \wedge y \in n(f)\} =$ { *a.(b), a.(c), a.(m), a.(k), a.(d), a.(a), b.(c), b.(m), b.(k), b.(d), b.(b), c.(a), c.(b), c.(m), c.(k), c.(d), c.(c), m.(a), m.(b), m.(c), m.(k), m.(d), m.(m), k.(a), k.(b), k.(c), k.(m), k.(d), k.(k), d.(a), d.(b), d.(c), d.(m), d.(k), d.(d)* }.

$\Sigma_0$ will be updated and $\Sigma_1$ will be created upon creating new names in the encoding. Let $m'$ be a new name in the encoding, then $\Sigma_0$ will be updated with this new name by calling function $update\Sigma(a, \Sigma_i)$ as follows:

$$\begin{aligned} update\Sigma(m', \Sigma_0) = \Sigma_0 \quad &\cup \quad \{m'.(c) \mid c \in channels(\Sigma_0)\} \\ &\cup \quad \{c.(m') \mid c \in channels(\Sigma_0)\} = \Sigma_1 \end{aligned}$$

Note that, function $update\Sigma(a, \Sigma_i)$ is called upon creating new names in the encoding of the input operator and the new ($\nu$) operator (see Definition 4.4).

- [**Calculating the process alphabet**] In $\mathcal{M}$CSP, as in CSP, a process alphabet is a permanent predefined property of this process and it consists of all the events that this process can perform [84]. The union of all alphabets in a system produce the $\Sigma$.

  In the encoding, the translated $\mathcal{M}$CSP process alphabet is obtained from the set of names of the $\pi$-calculus process as follows:

  Let $p$ be a $\pi$-calculus process and let $p'$ be the translation of $p$ in $\mathcal{M}$CSP, then $\alpha p'$ should be a subset of $\{a.(b) \mid a \in n(p) \wedge b \in channels(\Sigma)\}$.

  For instance, considering Example 4.4, if $f'$ is the translation of $f$ in $\mathcal{M}$CSP, then $\alpha f' = \{a.(y), b.(y) | y \in channels(\Sigma)\}$.

  In the rest of the chapter, we use $\alpha$ to refer to the $\mathcal{M}$CSP processes' alphabets.

**Mapping free names and bound names** The set $n(p)$ of names in the $\pi$-calculus process $p$ is partitioned into two sets: the set $fn(p)$ of *free names*, which can be communicated without restrictions; and the set $bn(p)$ of *bound names*, which cannot be communicated outside the process.

In $\mathcal{M}$CSP, as in CSP, all events are considered free, i.e. can be sent and received without restrictions. $\mathcal{M}$CSP does not admit bound names, so in the encoding we define a **set** $B$ of **bound names**, which is initially empty.

**Prefix mapping** Inspired by [123], we define below the encoding of $\pi$-calculus prefixes into $\mathcal{M}$CSP events:

- The input prefix $a(b)$ is translated as the $\mathcal{M}$CSP event $a?(b)$, where $b$ is an input variable.
- The output prefix $\bar{a}b$ is translated as the $\mathcal{M}$CSP event $\bar{a}!(b)$,where $b$ can be a variable or a value and the $\bar{\cdot}$, in the $\mathcal{M}$CSP output event, is defined as a **dual operator** for channel names in $\Sigma$, which works similarly to the $\pi$-calculus dual operator ($\overline{(a)} = \bar{a}$ and $\overline{(\bar{a})} = a$).

– Accordingly, we define a **renaming function (Dual Renaming)** $DR$ that maps every event in a process into its dual event using the dual operator.

$$DR(p) = p[\![\bar{x}/x | x \in \Sigma]\!]$$

This function is necessary to encode the parallel composition as will be shown later in this section.

Moreover, we update the definition of $\Sigma$ to be:

$$\Sigma := \Sigma \cup \{\bar{a}.(b) | a, b \in channels(\Sigma)\}$$

And we change the definition of the function $update\Sigma$ to be:

$$\begin{aligned} update\Sigma(a, \Sigma_i) = \Sigma_i \quad &\cup \quad \{a.(c) \mid c \in channels(\Sigma_i)\} \\ &\cup \quad \{\bar{a}.(c) \mid c \in channels(\Sigma_i)\} \\ &\cup \quad \{c.(a) \mid c \in channels(\Sigma_i)\} = \Sigma_{i+1} \end{aligned}$$

- The $\pi$-calculus silent action $\tau$ is encoded as the $\mathcal{M}$CSP event *tau*. In $\mathcal{M}$CSP, the silent action cannot be introduced directly by designers, so for the purpose of the encoding, we introduce ***tau*** as the observable version of $\tau$ and we define the set $\Sigma_\pi$ to be ($\Sigma_\pi := \Sigma \cup \{tau\}$). This event will be hidden later in the encoding using the *hiding* operator to produce the $\tau$ action.

For illustration consider the following example:

**Example 4.5.** Let $p \stackrel{\text{def}}{=} \tau.\bar{a}c.b(m).\mathbf{0}$ be a $\pi$-calculus process, and let $p'$ be the translated process of $p$ (see Definition 4.4 for the encoding), then $p'$ is defined as follows:

$$\begin{aligned} p' \;=\; & tau \rightarrow \bar{a}!(c) \rightarrow \texttt{ if } c \in B \texttt{ then } \big(b?(m) \rightarrow \\ & \quad \texttt{if } d \in B - \{c\} \texttt{ then } (alphaCOV.b!(\gamma(\delta)_1) \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(b?(m) \rightarrow \texttt{ if } d \in B \texttt{ then } (alphaCOV.b!(\gamma(\delta)_1) \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \end{aligned}$$

**Mapping transition labels** Our goal is to translate processes from the $\pi$-calculus to $\mathcal{M}$CSP in a way that preserves their behaviour. For this, we need to establish a mapping between the transition relation defined by the LTS for the $\pi$-calculus (see Figure 2.3) and the transition relation defined by the LTS of $\mathcal{M}$CSP (see Section 4.3.2). We start by defining a mapping between labels.

**Definition 4.3.** The mapping function$\widehat{\ }$from $\pi$-calculus labels into $\mathcal{M}$CSP labels is defined as follows:

If $a, b \in n(p)$ then:

- $\widehat{\bar{a}b} = \bar{a}.(b)$
- $\widehat{\bar{a}(b)} = \bar{a}.(b)$
- $\widehat{ab} = a.(b)$
- $\widehat{\tau} = \tau$

**Name space** In the $\pi$-calculus, an infinite set ***Names*** is available for defining communication channels and for creating fresh names. Inspired by the encoding in [123], we partition the set *Names* into two disjoint sets:

1. ***channels***$(\Sigma)$, which includes all the **known events** in the system;
2. $\delta = (\textbf{\textit{Names}} - \textbf{\textit{channels}}(\Sigma))$ which is an infinite set of names that can be used to create **fresh names**.

    - To update $\delta$ we define a function $\gamma(\delta)$ which returns an arbitrary element of $\delta$ and removes this element from $\delta$:

    $$\gamma(\delta) = (e, \delta - \{e\}) \quad where\ e \in \delta$$

    We use $\gamma(\delta)_1$ to denote **the new element**, i.e. the **first component** of the pair $\gamma(\delta)$, and $\gamma(\delta)_2$ to denote **the new** $\delta$, i.e. the **second component** of the pair $\gamma(\delta)$.

    - The function $\rho(\delta)$ **partitions the set** $\delta$ into two sets, we use $\rho(\delta)_1$ to denote the first set and $\rho(\delta)_2$ to denote the second set.

**$\alpha$-conversion** The function ***AC*** will be used to $\alpha$-convert names in a set to other names in case of name collision. We define this function as follows:

$$AC(SET, n, n') = SET[n/n']$$

where $SET$ denotes a set of names, and $n, n'$ denote the new and the old names.

We also add to $\Sigma_\pi$ a new event ***alphaCOV***, which is used to carry new names in case of inputting bound names. More precisely, we update $\Sigma_\pi$ to include this new event

$$\Sigma_\pi := \Sigma_\pi \cup \{alphaCOV\}$$

**Parallel composition association** We assume without loss of generality that in the $\pi$-calculus process to be translated, all parallel compositions associate to the left, i.e., they are written in the form: $(p|q)|r$. It is always possible to write a process in this way, using the structural congruence [127].

We are now ready to define the encoding $_{\mathcal{M}csp}[.]$ as a function from $\pi$-calculus processes to $\mathcal{M}$CSP processes. The encoding will be followed by a discussion and justification of our choices.

**Definition 4.4.** The encoding $_{\mathcal{M}csp}[.] : \pi \rightarrow \mathcal{M}$CSP is defined as follows:

$$_{\mathcal{M}csp}[p] = \lceil p \rfloor_{B,\Sigma_0,\delta} \setminus \{tau, alphaCOV\}$$

where $B$ is the set of bound names (initially empty), $\Sigma_0$ is the initial set $\Sigma$, $\delta$ is the space of fresh names, and $\lceil p \rfloor_{B,\Sigma,\delta}$ is inductively defined:

$$\begin{aligned}
\lceil \mathbf{0} \rfloor_{B,\Sigma,\delta} &= STOP \\
\lceil \tau.p \rfloor_{B,\Sigma,\delta} &= tau \rightarrow \lceil p \rfloor_{B,\Sigma,\delta} \\
\lceil a(b).p \rfloor_{B,\Sigma,\delta} &= a?(b) \rightarrow \texttt{if } b \in B \\
&\quad \texttt{then } \; alphaCOV.a!\gamma(\delta)_1 \rightarrow \\
&\quad\quad \lceil p\,[b/\gamma(\delta)_1] \rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2} \\
&\quad \texttt{else } \lceil p \rfloor_{B,update\Sigma(b,\Sigma),\delta}
\end{aligned}$$

$$
\begin{array}{rcl}
\lceil \bar{a}b.p \rfloor_{B,\Sigma,\delta} & = & \bar{a}!(b) \rightarrow \texttt{if } b \in B \texttt{ then } \lceil p \rfloor_{B-\{b\},\Sigma,\delta} \\
 & & \texttt{else } \lceil p \rfloor_{B,\Sigma,\delta} \\
\lceil p + q \rfloor_{B,\Sigma,\delta} & = & \lceil p \rfloor_{B,\Sigma,\delta} \,\Box\, \lceil q \rfloor_{B,\Sigma,\delta} \\
\lceil !p \rfloor_{B,\Sigma,\delta} & = & \|_{\alpha\lceil p \rfloor}^{\infty}{}_{i=1} \lceil p|p \rfloor_{B,\Sigma,\delta} \\
\lceil p|q \rfloor_{B,\Sigma,\delta} & = & \Big( \big( (\lceil p \rfloor_{B,\Sigma,\rho(\delta)_1} \,|||\, F_{pass}(FP)) \\
 & & \|_{\alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor)} \\
 & & (DR(\lceil q \rfloor_{B,\Sigma,\rho(\delta)_2}) \,|||\, F_{pass}(FP)) \big) \\
 & & \quad \backslash \alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor) \Big) [\![ x/\underline{x} | x \in FP ]\!] \,\|_{\Sigma_\pi}\, P_{reg}(B, \Sigma, \delta) \\
 & & \textbf{where } FP = (\alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor)) - B \\
\lceil (\nu c)\, p \rfloor_{B,\Sigma,\delta} & = & P_{reg}(\{\gamma(\delta)_1\}, S, D) \,\|_{\Sigma_\pi}\, \lceil\, p[c/\gamma(\delta)_1] \,\rfloor_{B \cup \{\gamma(\delta)_1\}, S, D} \\
 & & \textbf{where } D = \gamma(\delta)_2,\ S = update\Sigma(\gamma(\delta)_1, \Sigma)
\end{array}
$$

where the regulator process $P_{reg}(B, S, D)$, is defined as follows (adopted from [123]):

$$P_{reg}(B, S, D) = \Box \{\, a\_(b) \rightarrow Recur\_Clause \,|\, a.(b) \in S \wedge a \in (channel(S) - B)\}$$

and $\{\, a\_(b) \rightarrow Recur\_Clause \,|\, a.(b) \in S \wedge a \in (channel(S) - B)\}$ is enumerated as follows, $a.(b) \in S \wedge a \in (channel(S) - B)$ is a fixed condition in all clauses:

If the sent or received name is a free name then no change:

$$\{a?(b) \rightarrow P_{reg}(B, S, D) \,|\, b \notin B\} \,\Box\, \{\bar{a}!(b) \rightarrow P_{reg}(B, S, D) \,|\, b \notin B\}$$

If the sent name is a bound name then it is a case of scope extrusion, hence we omit the name from the bound names set:

$$\{\bar{a}!(b) \rightarrow P_{reg}(B - \{b\}, S, D) \,|\, b \in B\}$$

If the received name is a bound name then it is a case of name collision, hence we add the name to the free names and $\alpha$-convert the bound name:

$$\{a?(b) \rightarrow (alphaCOV.a?d \rightarrow P_{reg}(AC(B, b, d), update\Sigma(d, \Sigma), D)) \,|\, b \in B\}$$

While a process cannot emit unknown names (outside $fn \cup bn$) it can receive unknown names, if a new name has been created dynamically:

$$\{a?(b) \rightarrow P_{reg}(B \cup \{b\}, update\Sigma(b, \Sigma), D - \{b\}) | b \notin (channel(S) \cup B)\}$$

The regulator process $F_{pass}(FP)$, is defined as follows:

$$F_{pass}(FP) = \Box_{x:FP} \quad x \rightarrow \underline{x} \rightarrow F_{pass}(FP)$$

**Justification of the encoding choices and examples**

In the definition of the translation function, we notice the following features:

- [**Encoding the input and output prefixes**] Outputting a bound name ($\in B$) is considered as *scope extrusion*. Therefore, we remove the name from the set of bound names in the encoding ($B$).

  Inputting a bound name ($\in B$) into the process requires an $\alpha$-*conversion* for the internal name. Therefore, in this case we send a fresh name from the *regulator* process (explained later in this section) and we replace the name with the fresh name in the process, in the $\Sigma$ and in the set of the bound names $B$.

  Note that, in the encoding of input/ output prefixes, the `if-then-else` block is part of the translated process and not part of the encoding. This is because values of input/ output variables are not known during the encoding, and the `if-then-else` block ensures that the process will work correctly upon receiving/ retrieving values of input/ output variables.

  For illustration consider the following example:

  **Example 4.6.** Assume $B = \{m\}$ and we want to encode the $\pi$-calculus process $p \stackrel{\text{def}}{=} \bar{a}c.b(d).\mathbf{0}$

  According to Definition 4.4, $p$ should be encoded into $\mathcal{M}$CSP as follows:

Let $p'$ be the translated process of $p$, then $p'$ is defined as follows:

$$\begin{aligned} p' \;=\; & \bar{a}!(c) \rightarrow \texttt{ if } c \in \{m\} \texttt{ then } \big(b?(d) \rightarrow \\ & \quad \texttt{if } d \in \{\} \texttt{ then } (alphaCOV.b!(m') \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(b?(d) \rightarrow \texttt{ if } d \in \{m\} \texttt{ then } (alphaCOV.b!(m') \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \end{aligned}$$

In this example, the behaviour of the process depends on the value of $c$. Therefore, if $a$ sends a bound name (i.e. $c$ is substituted by $m$) then it is a case of a *scope extrusion*. For that, the sent name $m$ should be taken from the list of bound names and the second `if` will not consider $m$ as a bound name. However, if $a$ sends another name and $b$ receives a bound name (i.e. $d$ is substituted by $m$), then a new name will be issued (assume it is $m'$), the $m$ name in $B$ will be replaced by $m'$ (i.e. $B = \{m'\}$), and the $\Sigma$ will be updated by the new name $m'$.

- [**Encoding sums**] Encoding the $\tau$ as the explicit name *tau* makes the external choice of $\mathcal{M}$CSP expressive enough to encode the $\pi$-calculus choice.

  For illustration consider the following example:

  **Example 4.7.** Considering Example 4.4, and according to Definition 4.4, $f$ should be encoded into $\mathcal{M}$CSP as follows (recall that $B$ at the beginning of the encoding is empty):

  Let $f'$ be the translated process of $f$, then $f'$ is defined as follows:

  $f' = \lceil p \rfloor_{B,\Sigma,\delta} \,\Box\, \lceil q \rfloor_{B,\Sigma,\delta} \backslash \{tau, alphaCOV\}$

  Let $p'$ be the translated process of $p$ and $m'$ is the new name generated by $\gamma()$

function, then $p'$ is defined as follows:

$$\begin{aligned} p' \;=\; & \bar{a}!(c) \rightarrow \texttt{ if } c \in \{\} \texttt{ then } \big(tau \rightarrow b?(m) \rightarrow \\ & \quad \texttt{if } m \in \{\} \texttt{ then } (alphaCOV.b!(m') \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(tau \rightarrow b?(m) \rightarrow \texttt{ if } m \in \{\} \texttt{ then } (alphaCOV.b!(m') \rightarrow STOP) \\ & \texttt{ else } STOP\big) \end{aligned}$$

Let $q'$ be the translated process of $q$, then $q'$ is defined as follows:

$$\begin{aligned} q' \;=\; & a?(k) \rightarrow \texttt{ if } k \in \{\} \texttt{ then } \big(alphaCOV.a!(k') \rightarrow \bar{b}!(d) \rightarrow \\ & \quad \texttt{if } d \in \{\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(\bar{b}!(d) \rightarrow \texttt{ if } d \in \{\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \end{aligned}$$

- **[Encoding the replication operator]** Having the indexed version of the parallel composition indexed to $\infty$ makes it expressive enough for the $\pi$-calculus replication operator (!), i.e. unlimited number of copies of a process work in parallel. We render two processes at a time to correctly encode the parallel composition between the replicated process and any other processes (see Definition 4.4).

  For illustration consider Example 4.8. As a result of encoding the replicated process $\underset{\alpha\lceil p\rfloor}{\|}{}_{i=1}^{\infty} \lceil p|p\rfloor_{\{\},\Sigma,\rho(\delta)_1}$, in Example 4.8, we have the processes $\lceil p\rfloor_{\{\},\Sigma,\rho(\rho(\delta)_1)_1}$ and $DR(\lceil p\rfloor_{\{\},\Sigma,\rho(\rho(\delta)_1)_2})$ (the output of the *DR()* function) ready to communicate with the environment. In the example, the other process in the parallel composition is $DR(\lceil q\rfloor_{,\Sigma,\rho(\delta)_2})$. Thus, this process will be ready to communicate with $\lceil p\rfloor_{\{\},\Sigma,\rho(\rho(\delta)_1)_1}$. In the same time the two copies in the replicated process are able to communicate as expected (see Figure 2.3).

  **Example 4.8.** Let $f \stackrel{\text{def}}{=} (!p|q)$, and let $f'$ be the translated process of $f$, then

$f'$ is defined as follows:

$$\begin{aligned}\lceil !p|q\rfloor_{\{\},\Sigma,\delta} \;=\; & \Big(\big(\big(\underset{\alpha\lceil p\rfloor}{\|}\overset{\infty}{\underset{i=1}{}}\lceil p|p\rfloor_{\{\},\Sigma,\rho(\delta)_1} \,|||\, F_{pass}(FP)\big)\\ & \|_{\alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)}\\ & (DR(\lceil q\rfloor_{,\Sigma,\rho(\delta)_2}) \,|||\, F_{pass}(FP))\big)\\ & \backslash\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x\in FP]\!]\,\|_{\Sigma_\pi} P_{reg}(\{\},\Sigma,\delta)\end{aligned}$$

where $FP = \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)$ and

$$\begin{aligned}\lceil p|p\rfloor_{\{\},\Sigma,\rho(\delta)_1} \;=\; & \Big(\big(\big(\lceil p\rfloor_{\{\},\Sigma,\rho(\rho(\delta)_1)_1} \,|||\, F_{pass}(\alpha\lceil p\rfloor)\big)\\ & \|_{\alpha\lceil p\rfloor}\\ & (DR(\lceil p\rfloor_{\{\},\Sigma,\rho(\rho(\delta)_1)_2}) \,|||\, F_{pass}(\alpha\lceil p\rfloor))\big)\\ & \backslash\alpha\lceil p\rfloor\Big)[\![x/\underline{x}|x\in\alpha\lceil p\rfloor]\!]\,\|_{\Sigma_\pi} P_{reg}(\{\},\Sigma,\delta)\end{aligned}$$

- [**Encoding the parallel composition**] In the parallel composition, we use two auxiliary processes, namely *regulators* (inspired by [124, 123]). These regulators are composed in parallel with the main processes, and are used to allow some actions to be evaluated without being synchronised because of the main parallel composition.

  A regulator process is an external choice between a set of events which is often sent to the process as an argument [124]. Each choice will evaluate an event from the set and recursively call the regulator process again. The general form of a regulator is:

  $$reg(A) = \Box_{x:A}\, x \rightarrow reg(A)$$

  In the parallel composition we use two regulators as follows:

  - **Free pass regulator ($F_{pass}(FP)$):** In the $\pi$-calculus, if multiple processes are composed in parallel then any two processes (non-deterministically) can synchronise in shared events. To mimic this binary synchronisation of the $\pi$-calculus, we attach the regulator process $F_{pass}(FP)$ to each process in

the parallel composition. In this way, the process can evaluate the event or just pass it through the regulator.

More precisely, considering Example 4.9 below, if the output event $(\bar{a}!(b))$ is executed by $p'$ (i.e. the translated process of $p$), then this event will synchronise with the input event evaluated by either the $q'$ (i.e. the translated process of $q$), or by the $(F_{pass}(FP))$ regulator which is attached with $q'$. If the event is executed by the $q'$ then the synchronised event $(\bar{a}.(b)$, produced by rule (m-par1*) in Section 4.3.2) will be hidden afterwards because of the hiding operator which hides all shared events. If the event is executed by the $(F_{pass}(FP))$ regulator of $q'$ then the synchronised event $(\bar{a}.(b))$ will also be hidden afterwards because of the hiding operator, but the regulator will evaluate the same event renamed to $\underline{\bar{a}.(b)}$. The underline event will appear after the hiding operator, but will be renamed again to $\bar{a}.(b)$. Therefore, the double renaming allows the shared event to appear outside the parallel composition and, hence the process $p$, can synchronise with a third process outside this parallel composition.

Recall that the output event $(\bar{a}!(b))$ is executed by rule (ch-out1*) in Section 4.3.2 as $(\bar{a}.(b))$, input event $(\bar{a}?(c'))$ is executed by rule (ch-in1*) in Section 4.3.2 as $(\bar{a}.(b))$, and events in the $F_{pass}(FP)$ regulator are in the form as $\bar{a}.(b)$, where $\bar{a}.(b) \in \{|\bar{a}|\}$.

Note that, two regulators will never communicate, because both of them are processes of indexed external choices, and an external choice is resolved by an event offered by the environment not internally by the processes (External choice is explained in Section 2.4.3).

– **Bound names regulator ($P_{reg}(B, S, D)$):** This regulator explicitly implements the rules of bound names (passed as $B$ to the regulator) and free names (calculated as $\Sigma - B$). This process passes the free names without changing the communication, performs the scope extrusion by changing $B$, and $\alpha$-converting bound names in the case of name collisions. It is used within the parallel composition to catch the shared bound names of both processes in the parallel compositions. See the enumerated cases of process $P_{reg}(B, S, D)$ in Definition 4.4.

In addition to regulators the encoding of the parallel composition calls the function *DR()* (see the definition of *DR()* at the beginning of this section). In $\mathcal{M}$CSP, two input events or two output events can synchronise. To force input-output synchronization as in the $\pi$-calculus, we use the *dual operator* and the *DR()* function.

In the encoding, any input event is encoded to the same event ($a?x$ to $a?x$) whereas any output event is encoded to the dual event ($a!x$ to $\bar{a}!x$). Note that, in $\mathcal{M}$CSP, events which have the same name synchronise (i.e. $a$ with $a$ and $\bar{a}$ with $\bar{a}$). Therefore, in the parallel composition we compose a process with the *DR()* output of the other process (and its regulator), so *DR()* changes each event in the second process to its dual. Thus, outputs in the second process will be unbarred whereas input events will be barred. As a result, unbarred input events in the first process will synchronise with unbarred output event in the second process, and barred output in the first will synchronise with barred input in the second. Two inputs from the two processes will not synchronise as one is barred and the other is not.

Moreover, in the parallel composition, we split the space of fresh names $\delta$ into two spaces $\rho(\delta)_1$ and $\rho(\delta)_2$. This is necessary to ensure that the fresh names produced in the processes of the parallel composition are different.

Finally, the $\pi$-calculus assumes that processes in the parallel composition should synchronise on all shared events. Therefore, we design the interface set of the parallel composition to be the intersection of participants' alphabets. This is better than having all the $\Sigma$ to be the interface set as suggested in [123], because having all $\Sigma$ as interface set will block processes in the parallel composition if they try to evaluate an event that is not shared.

For illustration consider the following example (the $\pi$-calculus process in this example is adapted from [109])

**Example 4.9.** Let $f \stackrel{\text{def}}{=} p|q$, where $p \stackrel{\text{def}}{=} \bar{a}b.c(m).\mathbf{0}$

and $q \stackrel{\text{def}}{=} (\nu c)(a(c).\bar{b}d.\mathbf{0})$

be a $\pi$-calculus process, then $n(f) = n(p) \cup n(q) = \{a, b, c, m, d\}$.

Let $f'$ be the translated process of $f$ (see Definition 4.4 for the encoding), then $f'$ is defined as follows:

$f' = \lceil(\bar{a}b.c(m).\mathbf{0})|(\nu c)(a(c).\bar{b}d.\mathbf{0})\rfloor_{\{\},\Sigma,\delta}\backslash\{tau, alphaCOV\}$

$$\begin{aligned} f' \;=\; & \Big(\big((\lceil(\bar{a}b.c(m).\mathbf{0})\rfloor_{\{\},\Sigma,\rho(\delta)_1} \,|||\, F_{pass}(\{|\bar{a},b|\})\big) \\ & \|_{\{|\bar{a},b|\}} \\ & (DR(\lceil(\nu c)(a(c).\bar{b}d.\mathbf{0})\rfloor_{\{\},\Sigma,\rho(\delta)_2}) \,|||\, F_{pass}(\{|\bar{a},b|\}))\big) \\ & \backslash\{|\bar{a},b|\}\Big)[\![x/\underline{x}|x \in \{|\bar{a},b|\}]\!] \,\|_{\Sigma_\pi}\, P_{reg}(\{\},\Sigma,\delta) \end{aligned}$$

where $(\lceil(\bar{a}b.c(m).\mathbf{0})\rfloor_{\{\},\Sigma,\rho(\delta)_1} =$

$$\begin{aligned} & \bar{a}!(b) \rightarrow \texttt{ if } b \in \{\} \texttt{ then } \big(c?(m) \rightarrow \\ & \quad \texttt{if } m \in \{\} \texttt{ then } (alphaCOV.c!(m') \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(c?(m) \rightarrow \texttt{ if } m \in \{\} \texttt{ then } (alphaCOV.c!(m') \rightarrow STOP) \\ & \quad \texttt{else } STOP\big) \end{aligned}$$

and $\lceil(\nu c)(a(c).\bar{b}d.\mathbf{0})\rfloor_{\{\},\Sigma,\rho(\delta)_2} =$

$P_{reg}(\{c'\}, update\Sigma(c',\Sigma), \delta - \{c'\}) \,\|_{\Sigma_\pi}\, \lceil(a(c').\bar{b}d.\mathbf{0})\rfloor_{\{c'\},update\Sigma(c',\Sigma),\delta-\{c'\}}$

where $\lceil(a(c').\bar{b}d.\mathbf{0})\rfloor_{\{c'\},update\Sigma(c',\Sigma),\delta-\{c'\}} =$

$$\begin{aligned} & a?(c') \rightarrow \texttt{ if } c' \in \{c'\} \texttt{ then } \big(alphaCOV.a!(c'') \rightarrow \bar{b}!(d) \rightarrow \\ & \quad \texttt{if } d \in \{c''\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(\bar{b}!(d) \rightarrow \texttt{ if } d \in \{c'\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \end{aligned}$$

and $DR(\lceil(a(c').\bar{b}d.\mathbf{0})\rfloor_{\{c'\},update\Sigma(c',\Sigma),\delta-\{c'\}}) =$

$$\begin{aligned} & \bar{a}?(c') \rightarrow \texttt{ if } c' \in \{c'\} \texttt{ then } \big(alphaCOV.a!(c'') \rightarrow b!(d) \rightarrow \\ & \quad \texttt{if } d \in \{c''\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \\ & \texttt{else } \big(b!(d) \rightarrow \texttt{ if } d \in \{c'\} \texttt{ then } STOP \quad \texttt{else } STOP\big) \end{aligned}$$

Not that, the event $alphaCOV$ is in $\Sigma_\pi$ and not in $\Sigma$, therefore, $DR()$ function

does not change *alphaCOV* to its dual. Moreover, the event *alphaCOV* is not in *FP*, because *FP* is based on processes' alphabets which are based on $\Sigma$.

- [**Encoding the operator** $\nu$] In the case of encoding the creation a new bound name, we produce a new fresh name $\gamma(\delta)_1$ from the space $\delta$ by using the $\gamma()$ function (see the definition of $\gamma()$ at the beginning of this section), then we use this name in the process by attaching the regulator ($P_{reg}(B, S, D)$) which accepts this name as an argument and implements the rules which govern its freshness as shown in Definition 4.4.

  For illustration consider the encoding of process $q$ in Example 4.9.

### Properties of the Encoding

We now discuss the validity of the encoding by proving that there is an operational correspondence between a $\pi$-calculus process and its translated $\mathcal{M}$CSP process. This will be done by proving that the encoding is complete and sound according to Definition 4.2.

For the sake of clarity, when we write transitions for $\mathcal{M}$CSP processes we omit the store.

To facilitate the proofs in this section we first prove the following lemma:

**Lemma 4.1.** *Let $p$ be a $\mathcal{M}$CSP process and $P_{reg}$ be the regulator process given in Definition 4.4, then $(P_{reg}(\{\}, S, D) \parallel_{\Sigma_\pi} p) \sim p$, for any values $S, D$ used in Definition 4.4. Here $\sim$ denotes strong bisimilarity.*

*Proof.* We prove that if $B$ is empty in $P_{reg}(B, S, D)$ then for all $\sigma \in \Sigma$, if $p \xrightarrow{\sigma} p'$ then $P_{reg}(\{\}, S, D) \parallel_{\Sigma_\pi} p$ can only evaluate $\sigma$ and evolve to

$P_{reg}(\{\}, S, D) \parallel_{\Sigma_\pi} p'$.

By Definition 4.4, if $B$ is empty then:

$P_{reg}(\{\}, S, D) = (\Box \{ a?(b) \rightarrow P_{reg}(\{\}, S, D) \mid a.(b) \in S \wedge a \in channel(S) \}) \Box$
$(\Box \{ \bar{a}!(b) \rightarrow P_{reg}(\{\}, S, D) \mid a.(b) \in S \wedge a \in channel(S) \})$

where $S$ is the $\Sigma$ set or the updated version of the $\Sigma$ set (see Definition 4.4).

Hence, $P_{reg}(\{\}, S, D)$ cannot refuse any event in $\Sigma$ initiated by the environment and it cannot initiate any event by itself. As a result, if $P_{reg}(\{\}, S, D)$ is in parallel

with another process $p$, and $p$ synchronises on all events in $\Sigma$ with $P_{reg}(\{\}, S, D)$ then this parallel composition externally behaves like the process $p$.

Additionally, if $p \xrightarrow{\sigma} p'$, then this matches the process $p$ in the process $P_{reg}(\{\}, S, D) \parallel_{\Sigma_\pi} p$ doing the transition because of the previous justification. □

Additionally, we borrow Lemma 3.1 from Chapter 3. We recall the lemma below and for the proof refer to page 60.

**Lemma 4.2.** *If $q = p\backslash A$ then:*

1. *if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Leftrightarrow q \xrightarrow{\sigma} q'$, where $q' = p'\backslash A$.*
2. *if $\sigma \in A$ then $p \xrightarrow{\sigma} p' \Rightarrow q \xrightarrow{\tau} q'$, where $q' = p'\backslash A$. Also, $q \xrightarrow{\tau} q'$ implies either $p \xrightarrow{\sigma} p'$ for $\sigma \in A$ or $p \xrightarrow{\tau} p'$.*

**Theorem 4.1** (completeness)**.** *Let ${}_{\mathcal{M}csp}[] : \pi \to \mathcal{M}CSP$ be the encoding in Definition 4.4, then for all $\pi$-calculus process $f$ such that $f \xrightarrow{\sigma}_\pi f'$ it holds that ${}_{\mathcal{M}csp}[f] \stackrel{\widehat{\sigma}}{\Longrightarrow} {}_{\mathcal{M}csp}[f']$.*

*Proof.* The proof proceeds by induction on the $\pi$-calculus syntax (see Section 2.4.4). We will show that if a $\pi$ process performs a transition then its encoding process in $\mathcal{M}$CSP can do a corresponding weak transition. Note that, in the proofs, $\to$ denotes the $\mathcal{M}$CSP prefix operator and $\xrightarrow{\sigma}$ is a labelled transition. The cases are described as follows:

1. ---

$$\begin{aligned} f &= \mathbf{0} \\ {}_{\mathcal{M}CSP}[f] &= STOP \end{aligned}$$

This case holds trivially, because both processes have no transition.

2. ---

$$\begin{aligned} f &= \tau.p \\ {}_{\mathcal{M}CSP}[f] &= (tau \to \lceil p \rfloor_{B,\Sigma,\delta}) \backslash \{tau, alphaCOV\} \end{aligned}$$

The only possible transition for $(\tau.p)$ is: $\tau.p \xrightarrow{\tau}_{\pi} p$

by rule (TAU) in Figure 2.3.

This transition is matched by the encoded process in the following way:

$\mathcal{M}CSP[\tau.p] \xrightarrow{\tau} \mathcal{M}CSP\ [p]$

by rules: (prefix) and (hid2) in Figure 2.2.

3. ---

$$
\begin{aligned}
f &= a(b).p \\
\mathcal{M}CSP[f] &= (a?(b) \rightarrow \texttt{if } b \in B \texttt{ then } alphaCOV.a!\gamma(\delta)_1 \rightarrow \\
&\quad \lceil p\,[b/\gamma(\delta)_1] \rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2} \\
&\quad \texttt{else } \lceil p \rfloor_{B,update\Sigma(b,\Sigma),\delta}) \setminus \{tau, alphaCOV\}
\end{aligned}
$$

Recall that, $B$ is the set which holds the bound names (corresponding to the $bn$ set of the $\pi$ process).

The only possible transition for $(a(b).p)$ is:

$a(b).p \xrightarrow{ac}_{\pi} p[c/b]$ by rule (INP) in Figure 2.3.

This transition is matched by the encoded process in the following two ways:

(a) If $c \notin B$.

$\mathcal{M}CSP[a(b).p] \xrightarrow{a.(c)} \xrightarrow{\tau} \mathcal{M}CSP\ [p]$

by rules: (ch-in*) in Section 4.3.2, (hid1) in Figure 2.2, and (if1-3) in Section 2.4.3.

(b) If $c \in B$:

$\mathcal{M}CSP[a(b).p] \xrightarrow{a.(c)} \xrightarrow{\tau} \xrightarrow{\tau} \mathcal{M}CSP\ [p]$ by rules: (ch-in*) in Section 4.3.2, (hid1), (ch-out2*), (hid2) in Figure 2.2, and (if1-3) in Section 2.4.3.

where $a.(c) = \widehat{ac}$ according to Definition 4.3.

Note that, $_{\mathcal{M}CSP}[p]$ corresponds to $p[c/b]$, because $_{\mathcal{M}CSP}[p]$ is actually $(_{\mathcal{M}CSP}[p], \sigma[b \mapsto c])$, however, as aforementioned, in the proofs we omit stores from $\mathcal{M}$CSP transitions for simplicity.

4. ---

$$\begin{aligned} f &= \bar{a}b.p \\ \mathcal{MCSP}[f] &= (\bar{a}!(b) \rightarrow \texttt{if } b \in B \texttt{ then } \lceil p \rfloor_{B-\{b\},\Sigma,\delta} \texttt{ else} \\ &\quad \lceil p \rfloor_{B,\Sigma,\delta}) \setminus \{tau,\ alphaCOV\} \end{aligned}$$

The only possible transition for $(\bar{a}b.p)$ is: $\bar{a}b.p \xrightarrow{\bar{a}b}_{\pi} p$ by rule (OUT) in Figure 2.3.

Whether $b \in B$ or not, this transition is matched by the encoded process in the following way:

$\mathcal{MCSP}[\bar{a}b.p] \xrightarrow{\tau} \xrightarrow{\bar{a}.(b)} \xrightarrow{\tau}_{\mathcal{MCSP}} [p]$ by rules: (ch-out1), (ch-out2*) in Section 4.3.2, (hid1) in Figure 2.2, and (if1-3) in Section 2.4.3.

where $\bar{a}.(b) = \widehat{\bar{a}b}$ according to Definition 4.3.

5. ---

$$\begin{aligned} f &= p + q \\ \mathcal{MCSP}[f] &= (\lceil p \rfloor_{B,\Sigma,\delta} \Box \lceil q \rfloor_{B,\Sigma,\delta}) \setminus \{tau,\ alphaCOV\ \} \end{aligned}$$

The possible transition for $(p + q)$ is:

If $p \xrightarrow{\sigma}_{\pi} p'$ then $p + q \xrightarrow{\sigma}_{\pi} p'$ by rule (SUM-L) in Figure 2.3.

For the $\pi$-calculus sum, the choice is resolved by the first action done by either $p$ or $q$. Therefore, we consider in the following cases the first action only.

The sum transition is matched by the encoded process as follows:

(a) if $\sigma = a(b)$ or $\sigma = \bar{a}b$ then:
If $\mathcal{MCSP}[p] \xrightarrow{\tau}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}^{*}_{\mathcal{MCSP}} [p']$ then $\mathcal{MCSP}[p + q] \xrightarrow{\tau}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}^{*}_{\mathcal{MCSP}} [p']$ by rules: (exch1) and (hid1) in Figure 2.2.
We consider the action after the first sequence of $\tau$s because the $\mathcal{M}$CSP external choice can not be resolved by $\tau$.

(b) if $\sigma = \tau$ then:

If ${}_{\mathcal{M}CSP}[p] \xrightarrow{tau}{}^{*}_{\mathcal{M}CSP} [p']$ then ${}_{\mathcal{M}CSP}[p+q] \xrightarrow{\tau}{}^{*}_{\mathcal{M}CSP} [p']$ by rules: (exch1) and (hid2) in Figure 2.2.

Here, the $\pi$'s $\tau$ is encoded into the *tau* action therefore it is resolved by the $\mathcal{M}$CSP external choice.

6. ---

$$
\begin{aligned}
f &= (\nu c)\, p \\
{}_{\mathcal{M}CSP}[f] &= (P_{reg}(\{\gamma(\delta)_1\}, S, D) \parallel_{\Sigma_\pi} \lceil\, p[c/\gamma(\delta)_1] \,\rfloor_{B\cup\{\gamma(\delta)_1\},S,D}) \\
&\quad \backslash\{tau,\ alphaCOV\} \\
&\quad \textbf{where } D = \gamma(\delta)_2,\ S = update\Sigma(\gamma(\delta)_1, \Sigma)
\end{aligned}
$$

The possible transitions for $((\nu c)\, p)$ are as follows:

(a) If $p \xrightarrow{\sigma}_\pi p'$ then $((\nu c)\, p) \xrightarrow{\sigma}_\pi ((\nu c)\, p')$ if $c \notin n(\sigma)$

by rule (RES) in Figure 2.3.

This transition can take place only if $c$ is not communicated (i.e. $c \notin n(\sigma)$). Therefore, $\sigma = \tau$ or $\sigma$ is an input/output action and the communicated name is not bound (i.e. $c \notin bn(p)$).

This transition is matched by the encoded process in the following way:

We know by Lemma 4.2 that:

If ${}_{\mathcal{M}CSP}[p] \xrightarrow{\tau}{}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^{*}_{\mathcal{M}CSP} [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^{*} \lceil p' \rfloor_{B',\Sigma',\delta'}$

And by Definition 4.4:

If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^{*} \lceil p' \rfloor_{B',\Sigma',\delta'}$ then $(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma),$ $\gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil\, p[c/\gamma(\delta)_1] \,\rfloor_{B\cup\{\gamma(\delta)_1\},update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2})\backslash\{tau, alphaCOV\}$
$\xrightarrow{\tau}{}^{*} \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^{*} (P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi}$
$\lceil\, p' \,\rfloor_{B',\Sigma',\delta'})\backslash\{tau, alphaCOV\}$

by rules (exch1), (rec) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, which by Definition 4.4 is equivalent to: ${}_{\mathcal{M}CSP}[(\nu c)\, p']$

(b) If $p \xrightarrow{\bar{a}c}_\pi p'$ then $((\nu c)\, p) \xrightarrow{\bar{a}c}_\pi p'$ if $a \neq c$

by rule (OPEN) in Figure 2.3.

This transition can take place if $p$ evaluates an output action for a bound name, and it is matched by the encoded process in the following way:

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{}{}^{*} \xrightarrow{\bar{a}.(c)} \xrightarrow{}{}^{*}{}_{\mathcal{M}CSP}\ [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{}{}^{*} \xrightarrow{\bar{a}.(c)} \xrightarrow{}{}^{*}\ \lceil p' \rfloor_{B',\Sigma',\delta'}$.

And by Definition 4.4:

If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{}{}^{*} \xrightarrow{\bar{a}.(c)} \xrightarrow{}{}^{*}\ \lceil p' \rfloor_{B',\Sigma',\delta'}$ then $(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma),$ $\gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil\, p[c/\gamma(\delta)_1] \,\rfloor_{B\cup\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}) \backslash \{tau, alphaCOV\}$
$\xrightarrow{}{}^{*} \xrightarrow{\bar{a}.(c)} \xrightarrow{}{}^{*}\ (P_{reg}(\{\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi}$
$\lceil\, p' \,\rfloor_{B',\Sigma',\delta'}) \backslash \{tau, alphaCOV\}$

where $a \neq c$ and $c = \gamma(\delta)_1$

by using rules: (exch1), (rec) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2.

$(P_{reg}(\{\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil\, p' \,\rfloor_{B',\Sigma',\delta'}) \backslash \{tau, alphaCOV\}$
by Lemma 4.1 is equivalent to $(\lceil\, p' \,\rfloor_{B',\Sigma',\delta'}) \backslash \{tau, alphaCOV\}$.

By Definition 4.4: $(\lceil\, p' \,\rfloor_{B',\Sigma',\delta'}) \backslash \{tau, alphaCOV\}$ is $_{\mathcal{M}CSP}[p']$.

(c) If $p \xrightarrow{ac}_{\pi} p'$ then $((\nu c)\, p) \xrightarrow{ac}_{\pi} ((\nu c')\, p')$

by rule $\alpha$-conversion.

This transition can take place if $p$ evaluates an input action for a bound name, and it is matched by the encoded process in the following way:

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{}{}^{*} \xrightarrow{a.(c)} \xrightarrow{}{}^{*}{}_{\mathcal{M}CSP}\ [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{}{}^{*} \xrightarrow{a.(c)} \xrightarrow{}{}^{*}\ \lceil p' \rfloor_{B',\Sigma',\delta'}$.

And by Definition 4.4:

If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{}{}^{*} \xrightarrow{a.(c)} \xrightarrow{}{}^{*}\ \lceil p' \rfloor_{B',\Sigma',\delta'}$

then $(P_{reg}(B, \Sigma, \delta) \parallel_{\Sigma_\pi} \lceil\, p \,\rfloor_{B,\Sigma,\delta}) \backslash \{tau, alphaCOV\}$
$\xrightarrow{}{}^{*} \xrightarrow{a.(c)} (alphaCOV.a!c' \rightarrow (P_{reg}(AC(B, c, c'), update\Sigma(c', \Sigma), \delta - \{c'\})$
$\parallel_{\Sigma_\pi} \lceil\, p'[c/c'] \,\rfloor_{AC(B,c,c'), update\Sigma(c',\Sigma), \delta-\{c'\}}) \backslash \{tau, alphaCOV\}) \xrightarrow{}{}^{*} (P_{reg}(AC(B$
$, c, c'), update\Sigma(c', \Sigma), \delta - \{c'\}) \parallel_{\Sigma_\pi} \lceil\, p'[c/c'] \,\rfloor_{AC(B,c,c'), update\Sigma(c',\Sigma), \delta-\{c'\}})$
$\backslash \{tau, alphaCOV\}$

If $c \in B$, $a \neq c$, and assuming $\gamma(\delta)_1 = c'$.

By using rules: (ch-out2*) in Section 4.3.2, (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, (hid2), (hid1) in Figure 2.2.

Which by Definition 4.4 is equivalent to: $_{\mathcal{M}CSP}[(\nu c')p']$.

7. ---

$$
\begin{aligned}
f &= p|q \\
\mathcal{M}CSP[f] &= \Big(\Big(\big((\lceil p \rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP)\big) \\
&\quad \|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} \\
&\quad (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\big) \\
&\quad \backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big) [\![x/\underline{x}|x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\Big) \\
&\quad \backslash \{tau,\ alphaCOV\} \\
&\quad \textbf{where } FP = (\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)) - B
\end{aligned}
$$

The possible transitions for $(p|q)$ are as follows:

(a) If $p \xrightarrow{\sigma}_\pi p'$ then $(p|q) \xrightarrow{\sigma}_\pi (p'|q)$ by rule (PAR-L) in Figure 2.3.
This transition takes place if $bn(\sigma) \cap fn(q) = \emptyset$, i.e. $\sigma = \tau$ or $\sigma$ can be input/ output action that does not communicate a bound name which is considered free in the second process $q$, so $\sigma$ can be an unshared input/output action or shared input/output action but not synchronising on this action with $q$.
This transition is matched by the encoded process in the following way:
We know by Lemma 4.2 that:
If $_{\mathcal{M}CSP}[p] \xrightarrow{\tau}{}^* \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^*_{\mathcal{M}CSP} [p']$ then $\lceil p\rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^* \lceil p'\rfloor_{B',\Sigma',\delta'}$.
And by Definition 4.4:
If $\lceil p\rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^* \lceil p'\rfloor_{B',\Sigma',\delta'}$
then $\Big(\big((\lceil p\rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP)\big) \|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\big) \backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big) [\![x/\underline{x}|x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)$
$\backslash \{tau, alphaCOV\} \xrightarrow{\tau}{}^* \xrightarrow{\hat{\sigma}} \xrightarrow{\tau}{}^* \Big(\big((\lceil p'\rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP)\big)$
$\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\big) \backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big) [\![x/\underline{x}|x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)$
$\backslash \{tau, alphaCOV\}$.

This is achievable by using rules: (intv1) in Figure 2.2, (par1) in Section 4.3.2, (hid1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, if $\hat{\sigma} = a.(b)$ and $a.(b)$ is not a shared action.

Or, by using rules: (intv1) in Figure 2.2, (m-par1*) in Section 4.3.2, (exch1), (intv1), (rem1), (hid1), (rem1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, if $\hat{\sigma} = a.(b)$ and $a.(b)$ is a shared action.

Or, by using rules: (intv1) in Figure 2.2, (par1) in Section 4.3.2, (hid1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid2) in Figure 2.2, if $\hat{\sigma} = \tau$.

Which by Definition 4.4 is equivalent to: $_{\mathcal{M}CSP}[p'|q]$, because the sent name is not bound, and hence it is not included in $B$.

(b) If $p \xrightarrow{\bar{a}b}_\pi p'$ and $q \xrightarrow{ab}_\pi q'$ then $(p|q) \xrightarrow{\tau}_\pi (p'|q')$ by rule (COMM-L) in Figure 2.3.

This includes the case where $p$ and $q$ evaluate then synchronise on input/output actions.

This transition is matched by the encoded process in the following way:

We know that Lemma 4.2:

If $_{\mathcal{M}CSP}[p] \xrightarrow{\tau}{}^* \xrightarrow{\bar{a}.(b)} \xrightarrow{\tau}{}^*_{\mathcal{M}CSP} [p']$

then $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{\bar{a}.(b)} \xrightarrow{\tau}{}^* \lceil p' \rfloor_{B',\Sigma',\delta'}$,

and if $_{\mathcal{M}CSP}[q] \xrightarrow{\tau}{}^* \xrightarrow{a.(b)} \xrightarrow{\tau}{}^*_{\mathcal{M}CSP} [q']$

then $\lceil q \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{a.(b)} \xrightarrow{\tau}{}^* \lceil q' \rfloor_{B',\Sigma',\delta'}$.

And by Definition 4.4:

If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{\bar{a}.(b)} \xrightarrow{\tau}{}^* \lceil p' \rfloor_{B',\Sigma',\delta'}$ and $\lceil q \rfloor_{B,\Sigma,\delta} \xrightarrow{\tau}{}^* \xrightarrow{a.(b)} \xrightarrow{\tau}{}^*$
$\lceil q' \rfloor_{B',\Sigma',\delta'}$

then $\Big( \big( (\lceil p \rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP))$
$\|_{\alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor)} (DR(\lceil q \rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP)) \big)$
$\backslash \alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor) \Big) [\![ x/\underline{x} | x \in FP ]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta) \backslash \{tau, alphaCOV\}$
$\xrightarrow{\tau}{}^* \Big( \big( (\lceil p' \rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP))$
$\|_{\alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor)} (DR(\lceil q' \rfloor_{B',\Sigma',\rho(\delta)'_2}) ||| F_{pass}(FP)) \big)$
$\backslash \alpha\lceil p \rfloor \cap DR(\alpha\lceil q \rfloor) \Big) [\![ x/\underline{x} | x \in FP ]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta) \backslash \{tau, alphaCOV\}$.

By using rules: (intv1) in Figure 2.2, (m-par1*) in Section 4.3.2, (intv1), (hid2), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, where $a \in \alpha\lceil p \rfloor \cap \alpha\lceil q \rfloor$.

Which by Definition 4.4 is equivalent to: $_{\mathcal{M}CSP}[p'|q']$, because the sent name is not bound therefore it is not included in $B$.

(c) If $p \xrightarrow{\bar{a}(b)}_{\pi} p'$ and $q \xrightarrow{ab}_{\pi} q'$ then $(p|q) \xrightarrow{\tau}_{\pi} (\nu\, b)\,(p'|q')$ `if` $b \notin fn(q)$ by rule (CLOSE-L) in Figure 2.3.

This case is similar to case (b), except that in case (b) the transmitted name $b$ is not included in the set of bound names $B$ of the process, whereas in this case the transmitted name is included in the set of bound names $B$ of the process. In the encoding when the bound name is firstly encountered, the set $B$ is updated with a fresh name and sent to a dynamically created regulator which then is attached to this process.

In the current case, outputting a bound name outside the process will take the bound name out from the set of bound names. However, in the encoding of the parallel composition a bound name regulator is attached. In this regulator the transmission of this bound name is considered as a new name ($\notin \Sigma$). Therefore, the regulator will update the bound names set, the $\Sigma$, and the name space set $\delta$ (see the definition of $P_{reg}(\{b\}, \Sigma, \delta)$ regulator in Definition 4.4).

Thus,

$\Big(\big((P_{reg}(\{b\}, \Sigma, \delta) \|_{\Sigma_\pi} \lceil p \rfloor_{B,\Sigma,\rho(\delta)_1}) \,|||\, F_{pass}(FP)\big)$
$\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)}\ (DR(\lceil q \rfloor_{B,\Sigma,\rho(\delta)_2}) \,|||\, F_{pass}(FP))\big)$
$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big) [\![x/\underline{x} | x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(\{\}, \Sigma, \delta) \backslash \{tau, alphaCOV\}$
$\xrightarrow{\tau}{}^{*} \Big(\big(\big((P_{reg}(\{\}, \Sigma, \delta)(\|_{\Sigma_\pi} \lceil p' \rfloor_{B',\Sigma',\rho(\delta)'_1} \,|||\, F_{pass}(FP))$
$\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)}\ (DR(\lceil q' \rfloor_{B',\Sigma',\rho(\delta)'_2}) \,|||\, F_{pass}(FP))\big)$
$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big) [\![x/\underline{x} | x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B \cup \{b\}, update\Sigma(b, \Sigma), \delta - \{b\})$
$\backslash \{tau, alphaCOV\}$.

8. ---

$$\begin{aligned} f &= !p \\ \mathcal{M}CSP[f] &= (\underset{\alpha\lceil p\rfloor}{\|}\overset{\infty}{\underset{i=1}{}} \lceil p|p\rfloor_{B,\Sigma,\delta})\backslash\{tau, alphaCOV\} \end{aligned}$$

The possible transitions for $!p$ which are (REP-ACT), (REP-COMM), and (REP-CLOSE) in Figure 2.3, can be achieved as in cases 7.(a), 7.(b), and 7.(c) respectively. More precisely, rule (par-inf) will be used to produce the copy and cases 7.(a), 7.(b), and 7.(c) explain how in this copy the communications are achieved.

This completes the proof. □

**Theorem 4.2** (Soundness)**.** *Let $_{\mathcal{M}csp}[] : \pi \to \mathcal{M}CSP$ be the encoding in Definition 4.4, then for all $\pi$-calculus process $f$, such that $_{\mathcal{M}csp}[f] \xrightarrow{\sigma} T$, it holds that $f \xrightarrow{\sigma_1}_{\pi} f'$ and $T \overset{\rho}{\Longrightarrow}_{\mathcal{M}csp} [f']$, where in the sequence of transitions $\sigma\rho$ there is exactly one transition that corresponds to $\sigma_1$ and possibly some $\tau$ transitions.*

*Proof.* The proof proceeds by examining each case of the encoding in Definition 4.4 as follows (Note that, in the proofs, $\to$ denotes the $\mathcal{M}$CSP prefix operator and $\xrightarrow{\sigma}$ is a labelled transition):

1. ---

$$\begin{aligned} f &= \mathbf{0} \\ \mathcal{M}CSP[f] &= STOP \end{aligned}$$

This case holds trivially, because both processes have no transition.

2. ---

$$\begin{aligned} f &= \tau.p \\ \mathcal{M}CSP[f] &= (tau \to \lceil p\rfloor_{B,\Sigma,\delta})\backslash\{tau,alphaCOV\} \end{aligned}$$

If $_{\mathcal{M}CSP}[\tau.p] \xrightarrow{\sigma} T$, then $T =_{\mathcal{M}CSP} [p]$ and $\sigma = \tau$ , because the only possible transition for $(_{\mathcal{M}CSP}[\tau.p])$ is:

$_{\mathcal{M}CSP}[\tau.p] \xrightarrow{\tau}_{\mathcal{M}CSP} [p]$ by rules: (prefix) and (hid2) in Figure 2.2.

Then there is a $\pi$ transition:

$\tau.p \xrightarrow{\tau}_{\pi} p$ by rule (TAU) in Figure 2.3.

The encoding of $p$ is $_{\mathcal{M}CSP}[p]$ and $T =_{\mathcal{M}CSP} [p]$, therefore, $T \Rightarrow_{\mathcal{M}CSP} [p]$.

3. ---

$$
\begin{aligned}
f &= a(b).p \\
{}_{\mathcal{M}CSP}[f] &= (a?(b) \rightarrow \texttt{if } b \in B \texttt{ then } alphaCOV.a!\gamma(\delta)_1 \rightarrow \\
&\quad \lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2} \\
&\quad \texttt{else } \lceil p \rfloor_{B,update\Sigma(b,\Sigma),\delta}) \setminus \{tau, alphaCOV\}
\end{aligned}
$$

Recall that, $B$ is the set which holds the bound names (corresponding to the $bn$ set of the $\pi$ process).

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{\sigma}_{\mathcal{M}CSP} [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \setminus \{tau, alphaCOV\} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'} \setminus \{tau, alphaCOV\}$.

Therefore, if

$_{\mathcal{M}CSP}[a(b).p] \xrightarrow{\sigma} T$, then

$T = (\ \texttt{if}\ \ (c \in B)\ \ \texttt{then}\ \ alphaCOV.a!\gamma(\delta)_1 \rightarrow \lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}\ \ \texttt{else}\ \ \lceil p \rfloor_{B,update\Sigma(c,\Sigma),\delta}) \setminus \{tau, alphaCOV\}$

and $\sigma = a.(c)$, because the possible transition for $(_{\mathcal{M}CSP}[a(b).p])$ is:

$\big(a?(b) \rightarrow\ \texttt{if}\ \ (b \in B)\ \ \texttt{then}\ \ (alphaCOV.a!\gamma(\delta)_1 \rightarrow \lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2})\ \ \texttt{else}\ \ \lceil p \rfloor_{B,update\Sigma(b,\Sigma),\delta}\big) \setminus \{tau, alphaCOV\} \xrightarrow{a.(c)} \big(\ \texttt{if}\ \ (c \in B)\ \ \texttt{then}\ \ alphaCOV.a!\gamma(\delta)_1 \rightarrow \lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}\ \ \texttt{else}\ \ \lceil p \rfloor_{B,update\Sigma(c,\Sigma),\delta}\big) \setminus \{tau, alphaCOV\}$

by rules: (ch-in*) in Section 4.3.2 and (hid1) in Figure 2.2.

Then there is a $\pi$-transition: $a(b).p \xrightarrow{ac}_{\pi} p[c/b]$ by rule (INP) in Figure 2.3.

We can get from $T$ to $_{\mathcal{M}CSP}[p[c/b]]$ in the following two ways:

(a) If $c \notin B$, then

$\big($ `if` $(c \in B)$ `then` $alphaCOV.a!\gamma(\delta)_1 \rightarrow$
$\lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}$ `else` $\lceil p \rfloor_{B,update\Sigma(c,\Sigma),\delta}\big)$
$\backslash\{tau, alphaCOV\} \xrightarrow{\tau} \lceil p \rfloor_{B,update\Sigma(c,\Sigma),\delta} \backslash\{tau, alphaCOV\}$
by (if1-3) in Section 2.4.3.

(b) If $c \in B$, then

$\big($ `if` $(c \in B)$ `then` $alphaCOV.a!\gamma(\delta)_1 \rightarrow$
$\lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}$ `else` $\lceil p \rfloor_{B,update\Sigma(c,\Sigma),\delta}\big)$
$\backslash\{tau, alphaCOV\} \xrightarrow{\tau} \big(alphaCOV.a!\gamma(\delta)_1 \rightarrow$
$\lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}\big) \xrightarrow{\tau}$
$\lceil p\,[b/\gamma(\delta)_1]\,\rfloor_{AC(B,b,\gamma(\delta)_1),update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}$
by (if1-3) in Section 2.4.3 and the rules: (ch-out2*), (hid2) in Figure 2.2.

Therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p]$.

Note that, $_{\mathcal{M}CSP}[p]$ corresponds to $p[c/b]$, because $_{\mathcal{M}CSP}[p]$ is actually $(_{\mathcal{M}CSP}[p], \sigma[b \mapsto c])$ ($\sigma$ here is the store name, see Section 4.3.2), however, as aforementioned, in the proofs we omit stores from $\mathcal{M}$CSP transitions for simplicity.

Here, $\sigma = a.(c)$, $\rho = \tau$ and $\sigma_1 = ac$, thus, only $a.(c)$ corresponds to $ac$.

4. ---

$$
\begin{aligned}
f &= \bar{a}b.p \\
_{\mathcal{M}CSP}[f] &= (\bar{a}!(b) \rightarrow \texttt{if } b \in B \texttt{ then } \lceil p \rfloor_{B-\{b\},\Sigma,\delta} \texttt{ else} \\
&\quad \lceil p \rfloor_{B,\Sigma,\delta}) \backslash\{tau,\ alphaCOV\}
\end{aligned}
$$

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{\sigma} _{\mathcal{M}CSP}[p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \backslash\{tau, alphaCOV\} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$ $\backslash\{tau, alphaCOV\}$.

Therefore, if

$_{\mathcal{M}CSP}[\bar{a}b.p] \xrightarrow{\sigma} T$, then $T$ can be one of the following two cases:

(a) If $b$ is a variable,

then $T = \big(\bar{a}!(c) \rightarrow$ `if` $(c \in B)$ `then` $\lceil p \rfloor_{B-\{c\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\}$ and $\sigma = \tau$, because the possible transition for $(_{\mathcal{M}CSP}[\bar{a}b.p])$ is:

$\big(\bar{a}!(b) \rightarrow$ `if` $(b \in B)$ `then` $\lceil p \rfloor_{B-\{b\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\} \xrightarrow{\tau} \big(\bar{a}!(c) \rightarrow$ `if` $(c \in B)$ `then` $\lceil p \rfloor_{B-\{c\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big) \backslash\{tau, alphaCOV\}$.

By rules: (ch-out1) in Section 4.3.2 and (hid1) in Figure 2.2, where $c$ is the value of the variable $b$.

Then there is a $\pi$-transition:

$\bar{a}c.p \xrightarrow{\bar{a}c}_{\pi} p$ by rule (OUT), Figure 2.3.

We can get from $T$ to $_{\mathcal{M}CSP}[p]$ by the following transitions:

$\big(\bar{a}!(c) \rightarrow$ `if` $(c \in B)$ `then` $\lceil p \rfloor_{B-\{c\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\} \xrightarrow{\bar{a}.(c)} \big($ `if` $(c \in B)$ `then` $\lceil p \rfloor_{B-\{c\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\} \xrightarrow{\tau} (\lceil p \rfloor_{B',\Sigma,\delta}) \backslash\{tau, alphaCOV\}$.

By rules: (ch-out2*) in Section 4.3.2, (hid1) in Figure 2.2 and (if1-3) in Section 2.4.3, whether $b \in B$ or not.

Therefore, $T \overset{\bar{a}.(c)}{\Longrightarrow}_{\mathcal{M}CSP} [p]$.

Here, $\sigma = \tau$, $\sigma_1 = \bar{a}c$, and $\rho = \bar{a}.(c)$.

(b) If $b$ is a value,

then $T = \big($ `if` $(b \in B)$ `then` $\lceil p \rfloor_{B-\{b\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\}$ and $\sigma = \bar{a}!(b)$, because the possible transition for $(_{\mathcal{M}CSP}[\bar{a}b.p])$ is:

$\big(\bar{a}!(b) \rightarrow$ `if` $(b \in B)$ `then` $\lceil p \rfloor_{B-\{b\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\} \xrightarrow{\bar{a}.(b)} \big($ `if` $(b \in B)$ `then` $\lceil p \rfloor_{B-\{b\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta}\big)$ $\backslash\{tau, alphaCOV\}$.

By rules: (ch-out2*) in Section 4.3.2 and (hid1) in Figure 2.2.

Then there is a $\pi$ transition:

$\bar{a}b.p \xrightarrow{\bar{a}b}_{\pi} p$ by rule (OUT), Figure 2.3.

We can get from $T$ to $_{\mathcal{M}CSP}[p]$ by the following transition:
$\big($ `if` $(b \in B)$ `then` $\lceil p \rfloor_{B-\{b\},\Sigma,\delta}$ `else` $\lceil p \rfloor_{B,\Sigma,\delta})$
$\backslash\{tau, alphaCOV\} \xrightarrow{\tau} (\lceil p \rfloor_{B',\Sigma,\delta}\big)$
$\backslash\{tau, alphaCOV\}$, by (if1-3) in Section 2.4.3, whether $b \in B$ or not.
Therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p]$
Here, $\sigma = \bar{a}.(b)$, $\sigma_1 = \bar{a}b$, and $\rho = \tau$.

5. ---

$$\begin{aligned} f &= p + q \\ {}_{\mathcal{M}CSP}[f] &= (\lceil p \rfloor_{B,\Sigma,\delta} \Box \lceil q \rfloor_{B,\Sigma,\delta}) \backslash\{tau,\ alphaCOV\ \} \end{aligned}$$

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{\sigma}_{\mathcal{M}CSP} [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \backslash\{tau, alphaCOV\} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$ $\backslash\{tau, alphaCOV\}$.

Therefore, if

$_{\mathcal{M}CSP}[p + q] \xrightarrow{\sigma} T$, then $T$ can be one of the following two cases:

(a) If $(\lceil p \rfloor_{B,\Sigma,\delta}) \xrightarrow{\sigma} (\lceil p' \rfloor_{B,\Sigma,\delta})$ and $\sigma = a.(b)$ or $\sigma = \bar{a}.(b)$ then
$T = (\lceil p' \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\}$, because the possible transition for $(_{\mathcal{M}CSP}[p + q])$ is:
$(\lceil p \rfloor_{B,\Sigma,\delta} \Box \lceil q \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\} \xrightarrow{\sigma} (\lceil p' \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\}$
by using rules: (exch1) and (hid1) in Figure 2.2.
Then there is a $\pi$-transition:
If $p \xrightarrow{\sigma_1}_\pi p'$ then $p + q \xrightarrow{\sigma_1}_\pi p'$ by rule (SUM-L), Figure 2.3.
The encoding of $p'$ is $_{\mathcal{M}CSP}[p']$ and $T =_{\mathcal{M}CSP} [p']$.
Therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

(b) If $(\lceil p \rfloor_{B,\Sigma,\delta}) \xrightarrow{\sigma} (\lceil p' \rfloor_{B,\Sigma,\delta})$ and $\sigma = tau$ then
$T = (\lceil p' \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\}$, because the possible transition for $(_{\mathcal{M}CSP}[p + q])$ is:
$(\lceil p \rfloor_{B,\Sigma,\delta} \Box \lceil q \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\} \xrightarrow{\sigma} (\lceil p' \rfloor_{B,\Sigma,\delta}) \backslash\{tau, alphaCOV\}$
by using rules: (exch1) and (hid2) in Figure 2.2.

Then there is a $\pi$-transition:
If $p \xrightarrow{\sigma_1}_\pi p'$ then $p + q \xrightarrow{\sigma_1}_\pi p'$ by rule (SUM-L), in Figure 2.3.
The encoding of $p'$ is ${}_{\mathcal{M}CSP}[p']$ and $T =_{\mathcal{M}CSP} [p']$.
Therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

6. ---

$$
\begin{aligned}
f &= (\nu c)\, p \\
{}_{\mathcal{M}CSP}[f] &= (P_{reg}(\{\gamma(\delta)_1\}, S, D) \parallel_{\Sigma_\pi} \lceil p[c/\gamma(\delta)_1] \rfloor_{B\cup\{\gamma(\delta)_1\},S,D}) \\
&\quad \backslash\{tau,\ alphaCOV\} \\
&\quad \textbf{where } D = \gamma(\delta)_2,\ S = update\Sigma(\gamma(\delta)_1, \Sigma)
\end{aligned}
$$

We know by Lemma 4.2 that:

If ${}_{\mathcal{M}CSP}[p] \xrightarrow{\sigma}_{\mathcal{M}CSP} [p']$ then $\lceil p \rfloor_{B,\Sigma,\delta} \backslash \{tau, alphaCOV\} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'} \backslash \{tau, alphaCOV\}$.

Therefore, if

${}_{\mathcal{M}CSP}[(\nu c)\, p] \xrightarrow{\sigma} T$, then $T$ can be one of the following three cases:

(a) If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$ and $\sigma = tau$, $\sigma = \bar{a}.(b)$ or $\sigma = a.(b)$ and $b \notin B$, then $T = \big(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil\, p' \,\rfloor_{B',\Sigma',\delta'}\big) \backslash \{tau, alphaCOV\}$ because the possible transition for $({}_{\mathcal{M}CSP}[(\nu c)\, p])$ in this case is:
$\big(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil p[c/\gamma(\delta)_1] \rfloor_{B\cup\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}\big) \backslash \{tau, alphaCOV\} \xrightarrow{\sigma} \big(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil\, p' \,\rfloor_{B',\Sigma',\delta'}\big) \backslash \{tau, alphaCOV\}$
This could be achieved by rules: (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, (hid1) in Figure 2.2.
Then there is a $\pi$ transition:
If $p \xrightarrow{\sigma_1}_\pi p'$ then $((\nu c)\, p) \xrightarrow{\sigma_1}_\pi ((\nu c)\, p')$ if $c \notin n(\sigma)$ by rule (RES) in Figure 2.3.
The encoding of $((\nu c)\, p')$ is ${}_{\mathcal{M}CSP}[((\nu c)\, p')]$ and $T =_{\mathcal{M}CSP} [((\nu c)\, p')]$ by Definition 4.4, therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

(b) If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$ and $\sigma = \bar{a}.(c)$ and $c \in B$,
then $T = \big(P_{reg}(\{\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil p' \rfloor_{B',\Sigma',\delta'}\big)$
$\backslash\{tau, alphaCOV\}$ because the possible transition for $(_{\mathcal{M}CSP}[(\nu c)\, p])$ in this case is:
$\big(P_{reg}(\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi}$
$\lceil p[c/\gamma(\delta)_1] \rfloor_{B\cup\{\gamma(\delta)_1\}, update\Sigma(\gamma(\delta)_1,\Sigma),\gamma(\delta)_2}\big)\backslash\{tau, alphaCOV\}$
$\xrightarrow{\bar{a}.(c)} \big(P_{reg}(\{\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil p' \rfloor_{B',\Sigma',\delta'}\big)\backslash\{tau, alphaCOV\}$
If $a \neq c$ and $c = \gamma(\delta)_1$ by using rules: (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, (hid1) in Figure 2.2.
$\big(P_{reg}(\{\}, update\Sigma(\gamma(\delta)_1, \Sigma), \gamma(\delta)_2) \parallel_{\Sigma_\pi} \lceil p' \rfloor_{B',\Sigma',\delta'}\big)\backslash\{tau, alphaCOV\}$ is equivalent to $(\lceil p' \rfloor_{B',\Sigma',\delta'})\backslash\{tau, alphaCOV\}$ by Lemma 4.1.
$(\lceil p' \rfloor_{B',\Sigma',\delta'})\backslash\{tau, alphaCOV\}$ is $_{\mathcal{M}CSP}[p']$ by Definition 4.4.
Then there is a $\pi$-transition:
If $p \xrightarrow{\bar{a}c}_\pi p'$ then $((\nu c)\, p) \xrightarrow{\bar{a}c}_\pi p'$ and $a \neq c$ by rule (OPEN) in Figure 2.3.
The encoding of $p'$ is $_{\mathcal{M}CSP}[p']$ and $T =_{\mathcal{M}CSP} [p']$, therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

(c) If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$, $\sigma = a.(c)$, and $c \in B$,
then $T = (alphaCOV.a!c' \rightarrow (P_{reg}(AC(B, c, c'), update\Sigma(c', \Sigma), \delta - \{c'\}) \parallel_{\Sigma_\pi}$
$\lceil p'[c/c'] \rfloor_{B',\Sigma',\delta'})\backslash\{tau, alphaCOV\})$ because the possible transition for $(_{\mathcal{M}CSP}[(\nu c)\, p])$ in this case is:
$\big(P_{reg}(\{c\}, \Sigma, \delta) \parallel_{\Sigma_\pi} \lceil p \rfloor_{B,\Sigma,\delta}\big)$
$\backslash\{tau, alphaCOV\} \xrightarrow{a.(c)} \big(alphaCOV.a!c' \rightarrow (P_{reg}(AC(B, c, c'),$
$update\Sigma(c', \Sigma), \delta - \{c\}) \parallel_{\Sigma_\pi} \lceil p'[c/c'] \rfloor_{B',\Sigma',\delta'}\big)\backslash\{tau, alphaCOV\})$
If $a \neq c$ and assuming $\gamma(\delta)_1 = c$.
By using rules: (ch-in*) in Section 4.3.2, (if1-3) in Section 2.4.3, (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2.
Then there is a $\pi$-transition:
If $p \xrightarrow{ac}_\pi p'$ then $((\nu c)\, p) \xrightarrow{ac}_\pi ((\nu c')\, p')$ by rule (INP) in Figure 2.3 and ($\alpha$-conversion).
We can get from $T$ to $_{\mathcal{M}CSP}[(\nu c')\, p']$ in the following way:

$\Big(alphaCOV.a?c' \rightarrow (P_{reg}(AC(B,c,c'), update\Sigma(c',\Sigma), \delta - \{c'\})$
$\|_{\Sigma_\pi} \lceil\, p'[c/c'] \,\rfloor_{B',\Sigma',\delta'}\Big)\backslash\{tau, alphaCOV\}) \xrightarrow{\tau}$
$\Big(P_{reg}(AC(B,c',c), update\Sigma(c',\Sigma), \delta - \{c'\}) \,\|_{\Sigma_\pi} \lceil\, p'[c/c'] \,\rfloor_{B',\Sigma',\delta'}\Big)$
$\backslash\{tau, alphaCOV\}$

if $a \neq c$ and assuming $\gamma(\delta)_1 = c$.

By using rules: (m-par1*) in Section 4.3.2, (hid2), and (hid1) in Figure 2.2.

Which by Definition 4.4 is equivalent to: $_{\mathcal{M}CSP}[(\nu c')\, p']$.

7. ---

$$
\begin{aligned}
f &= p|q \\
\mathcal{M}CSP[f] &= \Big(\Big(((\lceil p \rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP)) \,\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} \\
&\quad (DR(\lceil q \rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\Big) \\
&\qquad \backslash\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \,\|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\Big) \\
&\quad \backslash\{tau,\ alphaCOV\} \\
&\quad \textbf{where } FP = (\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)) - B
\end{aligned}
$$

We know by Lemma 4.2 that:

If $_{\mathcal{M}CSP}[p] \xrightarrow{\sigma} {}_{\mathcal{M}CSP}\,[p']$

then $\lceil p \rfloor_{B,\Sigma,\delta}\backslash\{tau, alphaCOV\} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}\backslash\{tau, alphaCOV\}$.

Therefore, if

$_{\mathcal{M}CSP}[p|q] \xrightarrow{\sigma} T$, then $T$ can be one of the following three cases:

(a) If $\lceil p \rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p' \rfloor_{B',\Sigma',\delta'}$ and ($\sigma = \tau$, $\sigma = a.(b)$, or $\sigma = \bar{a}.(b)$),
then $T = \Big(((\lceil p' \rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP)) \,\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)}$
$(DR(\lceil q \rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\Big) \backslash\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!]$
$\|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\backslash\{tau, alphaCOV\}$
If ($b \notin B$ or ($b \in B$ and $b \notin \alpha\lceil q \rfloor$)), this means that the action is not carrying a bound name which appears free in the second process. These

actions include: not shared input/ output actions or shared input/ output actions but not synchronising with the other process.

Because the possible transition for $(_{\mathcal{M}CSP}\lceil p|q\rceil)$ in this case is:
$\Big(\big((\lceil p\rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP)) \parallel_{\alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2})$
$||| F_{pass}(FP))\big) \backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \parallel_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)$
$\backslash\{tau, alphaCOV\} \xrightarrow{\sigma} \Big(\big((\lceil p'\rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP))$
$\parallel_{\alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\big) \backslash \alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \parallel_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\backslash\{tau, alphaCOV\}$.

By rules: (intv1) in Figure 2.2, (par1) in Section 4.3.2, (hid1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, if $\sigma = a.(b)$ and $a.(b)$ is not shared.

Or, by rules: (intv1) in Figure 2.2, (m-par1*) in Section 4.3.2, (exch1), (intv1), (rem1), (hid1), (rem1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, if $\sigma = a.(b)$ and $a.(b)$ is shared.

Or, by rules: (intv1) in Figure 2.2, (par1) in Section 4.3.2, (hid1), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid2) in Figure 2.2, if $\sigma = \tau$.

Then there is a $\pi$-transition:

If $p \xrightarrow{\sigma_1}_\pi p'$ then $(p|q) \xrightarrow{\sigma_1}_\pi (p'|q)$ if $bn(\sigma) \cap fn(q) = \emptyset$ by rule (PAR-L) in Figure 2.3.

The encoding of $p'|q$ is $_{\mathcal{M}CSP}[p'|q]$ and $T =_{\mathcal{M}CSP} [p'|q]$ by Definition 4.4, therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

(b) If $\lceil p\rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p'\rfloor_{B',\Sigma',\delta'}$, $\lceil q\rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil q'\rfloor_{B',\Sigma',\delta'}$, and ($\sigma = a.(b)$ or $\sigma = \bar{a}.(b)$),
then $T = \Big(\big((\lceil p'\rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP))$
$\parallel_{\alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q'\rfloor_{B',\Sigma',\rho(\delta)'_2}) ||| F_{pass}(FP))\big)$
$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \parallel_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\backslash\{tau, alphaCOV\}$
If $b \notin B$, which means that $p$ and $q$ evaluate then synchronise input/output actions.

Because the possible transition for $(_{\mathcal{M}CSP}\lceil p|q\rceil)$ in this case is:
$\Big(\big((\lceil p\rfloor_{B,\Sigma,\rho(\delta)_1} ||| F_{pass}(FP))$
$\parallel_{\alpha\lceil p\rfloor\cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\big)$

$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\backslash\{tau, alphaCOV\}$
$\xrightarrow{\tau} \Big(\big((\lceil p'\rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP)\big)$
$\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q'\rfloor_{B',\Sigma',\rho(\delta)'_2}) ||| F_{pass}(FP))\Big)$
$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \|_{\Sigma_\pi} P_{reg}(B,\Sigma,\delta)\backslash\{tau, alphaCOV\}.$

By rules: (intv1) in Figure 2.2, (m-par1*) in Section 4.3.2, (intv1), (hid2), (exch1) in Figure 2.2, (m-par1*) in Section 4.3.2, and (hid1) in Figure 2.2, where $a \in \alpha\lceil p\rfloor \cap \alpha\lceil q\rfloor$.

Then there is a $\pi$-transition:

If $p \xrightarrow{\bar{a}b}_\pi p'$ and $q \xrightarrow{ab}_\pi q'$ then $(p|q) \xrightarrow{\tau}_\pi (p'|q')$ by rule (COMM-L) in Figure 2.3.

The encoding of $p'|q'$ is $_{\mathcal{M}CSP}[p'|q']$ and $T =_{\mathcal{M}CSP} [p'|q']$ by Definition 4.4, therefore, $T \Longrightarrow_{\mathcal{M}CSP} [p']$.

(c) If $\lceil p\rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil p'\rfloor_{B',\Sigma',\delta'}$, $\lceil q\rfloor_{B,\Sigma,\delta} \xrightarrow{\sigma} \lceil q'\rfloor_{B',\Sigma',\delta'}$ and $b \in B$, where $\sigma = a.(b)$ or $\sigma = \bar{a}.(b)$.

In this case, $p$ and $q$ evaluate then synchronise input/output action and this action is bound. This case is similar to case (b), except that in case (b) the transmitted name $b$ is not included in the set of bound names $B$ of the process, whereas in this case the transmitted name is included in the set of bound names $B$ of the process. In the encoding when the bound name is firstly encountered, the set $B$ is updated with a fresh name and sent to a dynamically created regulator which then is attached to this process.

In the current case, outputting a bound name outside the process will take the bound name out from the set of bound names. However, in the encoding of the parallel composition a bound name regulator is attached. In this regulator the transmission of this bound name is considered as a new name ($\notin \Sigma$). Therefore, the regulator will update the bound names set, the $\Sigma$, and the name space set $\delta$ (see the definition of $P_{reg}(\{b\},\Sigma,\delta)$ regulator in Definition 4.4).

Thus,

$\Big(\big((P_{reg}(\{b\},\Sigma,\delta)\|_{\Sigma_\pi}\lceil p\rfloor_{B,\Sigma,\rho(\delta)_1}) ||| F_{pass}(FP)\big)$
$\|_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q\rfloor_{B,\Sigma,\rho(\delta)_2}) ||| F_{pass}(FP))\Big)$

$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \parallel_{\Sigma_\pi} P_{reg}(\{\}, \Sigma, \delta)\backslash\{tau, alphaCOV\}$
$\xrightarrow{\sigma} \Big(\big(\big((P_{reg}(\{\}, \Sigma, \delta)(\parallel_{\Sigma_\pi} \lceil p'\rfloor_{B',\Sigma',\rho(\delta)'_1} ||| F_{pass}(FP))$
$\parallel_{\alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)} (DR(\lceil q'\rfloor_{B',\Sigma',\rho(\delta)'_2}) ||| F_{pass}(FP))\big)$
$\backslash \alpha\lceil p\rfloor \cap DR(\alpha\lceil q\rfloor)\Big)[\![x/\underline{x}|x \in FP]\!] \parallel_{\Sigma_\pi} P_{reg}(B \cup \{b\}, update\Sigma(b, \Sigma), \delta - \{b\})$
$\backslash\{tau, alphaCOV\}$.

This is matched by the $\pi$ process transition:
If $p \xrightarrow{\bar{a}(b)}_\pi p'$ and $q \xrightarrow{ab}_\pi q'$ then $(p|q) \xrightarrow{\tau}_\pi \nu\, b\, (p'|q')$ `if` $b \notin fn(q)$ by rule (CLOSE-L) in Figure 2.3.

8. ---

$$\begin{aligned} f &= !p \\ \mathcal{M}CSP[f] &= (\underset{\alpha\lceil p\rfloor}{\parallel}{}_{i=1}^{\infty} \lceil p|p\rfloor_{B,\Sigma,\delta})\backslash\{tau, alphaCOV\} \end{aligned}$$

In this case, we use rule (par-inf) to produce a copy from the replicated process, i.e. $\lceil p|p\rfloor_{B,\Sigma,\delta}$. After that, we discuss the possible transitions for $\lceil p|p\rfloor_{B,\Sigma,\delta}$ as we did in cases 7.(a), 7.(b), and 7.(c), and these cases are matched by the $\pi$-calculus process in rules: (REP-ACT), (REP-COMM), and (REP-CLOSE), in Figure 2.3 respectively.

This completes the proof.

□

## 4.5 Conclusions and Related work

Mobility is a crucial feature in any process calculus proposed for modelling complex systems, since mobility primitives facilitate dynamic reconfigurations of systems. This can be achieved by allowing channel names to be passed around as a representation of a communication delegated from one process to another.

In this chapter, we enhance CSP by allowing mobile communications in addition to its standard mixed synchronous/ interleaving communications. More precisely,

we extend the parallel composition operator in order to permit channels to be carried around in the calculus and we update the parallel composition's interface set accordingly.

We argued in the chapter that our mobility model is expressive, that it allows channel names to be sent and released, and takes into account CSP's generalised version of parallel composition. However, as mentioned in the introduction, there are other calculi, in the literature, which incorporate mobility into CSP. Below we briefly discuss these models and then compare them to our model.

Occam-$\pi$ [140] is presented as a mobile version of occam (a concurrent programming language founded on CSP). Mobility is obtained in occam-$\pi$ by declaring communicated channels as a bundle of a fixed number of sub-channels with flexible ends. Mobile communications are employed by processes connecting and disconnecting from these ends.

Replacing channels in a process alphabet with a bundle of channels is not strictly increasing the communication capability at runtime, instead it offers a range of possible communications that can be adjusted at runtime. This solution is more suitable for programming languages as argued in [124].

Another mobility model has been proposed in [122], and has been adopted in (CSP $\|$ B) [134]. In this model, mobility is obtained by sending the right to use a channel instead of sending the channel name to avoid changing the actual processes' alphabets. This mobility model has been employed in Hoare's CSP [84], where the parallel composition's interface set is always equal to the intersection between participants' alphabets (called alphabetised in [124]).

Introducing rights in addition to using channels complicates the calculus and does not strictly add mobility where alphabets are still fixed and the rights to use channels are circulated. Additionally, this model is based on the alphabetised parallel composition, instead we use the more general version of CSP parallel composition, namely generalised parallel composition.

The model which is presented in [124], allows channels to be communicated (maximum one at a time) and the processes' alphabets are dynamically changed. In this model, channels are sent with marks $(-,+)$ to guide changes to processes alphabets, where $+$ adds the channel to the processes' alphabets and $-$ removes it. The alphabetised parallel composition is then used to ensure that the changes in a participant's

alphabet are included in communications every time. Using the alphabetised parallel composition means that any shared events should be synchronised. As a result, designers do not have the right to exclude any of these events from synchronising as they can do in the generalised version of the parallel composition [124].

In comparison to these models, our model allows channels to be communicated without conditions. In our model we use marks $+, -$ to guide changes in the interface set of the generalised parallel composition; processes' alphabets are dynamically changed accordingly. Relaxing the condition of synchronising on all shared events, provides designers with the choice to synchronise on events or not. Additionally, having this feature in our model allows processes to dynamically switch communications between synchronous and interleaving modes by changing the interface set, which to the best of our knowledge has not been developed in the other models.

Finally, we discuss the relationship between our work and a previous encoding of the $\pi$-calculus in CSP. Roscoe [123] studies the expressive power of CSP and describes an encoding of the $\pi$-calculus into CSP, thus defining a semantics for the $\pi$-calculus in CSPs models. In this encoding, the replication operator is restricted (due to the fact that CSP does not permit unguarded recursions), whereas we have extended CSP to be able to express infinite replication. One of the main challenges in an encoding of the $\pi$-calculus in CSP is dealing with fresh names; for this, a generalised relabelling operator is used in [123], which can be expressed in terms of standard CSP operators. The relabelling ensures that when two names appear in nested or separate scopes they are different, and avoids clashes between names used in a process and names generated by the environment (various mechanisms to manage the name space are described). Our management of names is inspired by this work, but instead of encoding $\alpha$-conversion by replacing the input name and update the environment accordingly, we do it like in the $\pi$-calculus by replacing the internal bound name (see Definition 4.4). Operational correspondence has not been proved in [123], instead, the CSP semantics of the $\pi$-calculus is used to define a notion of refinement for $\pi$-calculus processes. An implementation in the FDR is also discussed, for processes that satisfy certain conditions (the networks cannot grow unboundedly).

# 5

# Session-Based CSP (CSPs)

## 5.1 Introduction

In the Service Oriented Computing (SOC) paradigm, services can be concurrently invoked in many compositions by creating different instances of these services in each composition. Hence, it is important to guarantee that messages are routed to the right instance. In the web services standards [14, 15], this is achieved by using *correlation sets*, where predefined subsets of data parameters (e.g., userID) are used as explicit tags to avoid interference between instances' communications (e.g., the message "send.userID.cardN" between a client and a bank service will become "send.1232.cardN" at runtime for user 1232). Although correlation sets are adequately expressive to distinguish between communications from different instances of a service, it has been argued that using correlation sets might complicate static analysis because interactions rely on data values [30]. For instance, services may interfere with each other's interactions if (on purpose or by chance) the same value is used.

To overcome this situation, the notion of session has been proposed, where a data independent key – a session key – is used to bind a series of communications between two services. This session key could be implicit (generated by the runtime system) [35, 93, 36], or explicit (set by the designer) [53]. However, sessions in all these works can be established between only two parties. In [35, 93, 36], sub sessions are allowed, and invocations within the first session creates a sub-session. This limits the scope of the composition and requires the intervention of the system designer to explicitly circulate results between services in different sessions.

To facilitate multiparty sessions, new notions were formulated in [137, 45], called conversation endpoints and session access points, respectively. However, the burden of generating and maintaining multiparty session notions lies on the designer of the system, increasing the risk of errors. In [87, 28] multiparty sessions are also supported but sessions' participants should be explicitly indicated when sessions are created.

In this chapter, we propose a new process calculus, called *session-based CSP (CSPs)*, where multiparty sessions can be created, managed and terminated transparently without involving the designer. Our calculus is a session-based extension of Hoare's CSP [84]. CSP has been chosen as the foundation for our calculus due to its multi-way communication model. In CSP, all processes participating in a parallel composition synchronise on a predefined set of events.

CSPs extends the standard CSP syntax with new primitives, supported by operational semantics.

**Contributions of this chapter:**

1. We formally introduce multiparty sessions into CSP. Multiparty sessions are maintained in the semantics without the intervention of designers. This facilitates the natural evolution of service invocations without extra formal notions (such as endpoints or sites).

2. In CSPs, service compositions are started with two parties and grow by invocations. When invocations are made, designers have the choice of integrating the new service into the current session or starting a new subsession with the ability to communicate with the parent session.

3. In CSPs, communications within sessions can combine synchronous and interleaving modes. Although multicast communications and mixed mode communications are inherited from CSP, maintaining these communications solely within session boundaries and avoiding interference between sessions is not trivial. We show how this can be achieved transparently in CSPs.

4. In CSPs, we introduce new termination primitives to facilitate graceful termination within the hierarchy of sessions. Moreover, we give services in sessions the

choice to interrupt their executions and optionally start termination handlers if their siblings in a session have terminated.

5. In CSPs, multicast communications of CSP happen within multiparty sessions. Therefore, new scenarios like (WS-coordination [16]) are now possible in CSPs.

**Structure of this chapter :** Section 5.2 provides an informal description of CSPs. Section 5.3 presents the extended syntax and operational semantics of CSPs. Section 5.4 proves some CSPs properties. Finally, Section 5.5 concludes the chapter and discusses related works.

## 5.2 CSPs Model

In our model, labels are the semantic elements that are considered as the representation of session names (keys). Labels are like implicit tags (not visible to designers) for communications belonging to one session. In the literature, *sessions* are usually defined as a series of interactions between two entities, whereas *multiparty sessions* or conversations are interactions which might involve more than two entities. In this chapter, we use multiparty sessions and sessions interchangeably to denote a series of interactions between two or more entities. Sessions here are untimed and a session is active as long as the entities are communicating.

Before proceeding to the content of this section we define the following terms to facilitate the understanding of the CSPs model:

- *Published service:* is the service which offers a function.
- *Invocation service:* is the client partner which invokes a published service.
- *Invoked service:* is the published service which has been invoked in the current session.
- *Invocation:* is the mechanism by which an invocation service invokes a published service and they create a new session. If an invocation is made within a running session then a new sub-session will be created.

- *Join invocation:* is the mechanism by which an invocation service invokes a published service to join the current running session. Here, invocation services are called join invocation services.

### 5.2.1 Sessions

Essentially, CSPs session labels are inspired by the CSP labelling operator [84], where processes can be renamed by attaching the label name at the front of their event names. Formally, we define session labels as follows:

**Definition 5.1** (Labels)**.** If $p$ is a process then $l : p$ is the same process working under session $l$. If a process is participating in session $l$ then all its events will be temporarily renamed to have the label name as well. Thus, if $p$ is ready to engage in event $a$ then $l : p$ will be ready to engage in event $l.a$, which means that the process $l : p$ is performing the event $a$ under session $l$. We use $l, l_0, \ldots, l_n, l', l'', \ldots$ to denote labels. Labels, session keys, and session names are used interchangeably in this chapter to refer to session names (keys). We use the function *new* to produce a fresh label.

Labels are implicit tags so designers do not need to install, maintain, or terminate sessions. However, designers should be aware of their existence and should express their desire to let the system work under sessions by using SOC idioms. SOC idioms in CSPs are as follows:

- *Service publishing*: $N \Rightarrow p$ denotes publishing the service $p$ with the name $N$, so $N$ is the published name of the ordinary CSP process $p$.
- *Service invocation*: $N \Leftarrow \{q\}$ denotes the invocation of a published service with name $N$ by the ordinary CSP process $q$ (we call $q$ the *client protocol*).

When a published service and an invocation to this service are put in parallel, a session will be created and interactions will be tagged with this session name to avoid interference with other communications outside the session.

Sessions will be created using an algorithm illustrated by the following example:

Assume the designer wrote: $N \Rightarrow p \parallel N \Leftarrow \{q\}$

According to our model, a session will be created as follows:

1. The published service ($N \Rightarrow p$) will create a new label $l$ using the function $new$ ($(new)(N \Rightarrow p)$).

2. The published service will be ready to work under this session name ($l : p$).

3. The client $q$ which is invoking the published service $N$ ($N \Leftarrow \{q\}$) will be ready to work under any session offered by the published service.

4. If the published service is composed in parallel with a client which is invoking it, then the label which, in our semantics, represents the session name will be passed to the invocation.

Thus, $N \Rightarrow p \parallel N \Leftarrow \{q\}$ will be reduced, using our semantics, to $l : p \parallel l : q$. This will tag the communications between these two processes with a new label $l$, see Figure 5.1.NS1.

In the algorithm, the published service is the one who is responsible for creating labels. This complies with the real life situation where the server is the one who creates session keys, and most probably the current SOC system will use the underlying server-client network. In this way, the semantics for session creation facilitates smoother implementation and avoids ambiguity.

This scenario assumes that the published and the invocation services are not working under any current session. Therefore, their communication creates a new session. However, in realistic situations, these services might already be under a current active session. In the SOC paradigm, a working service might invoke another service (triggers the execution of this invoked service) to do a particular task for it, then the invocation service might share the result with other services in the same composition. Hence, the main context is the one where the invocation is made. Therefore, if the invocation is made within an active session, this session label should be kept for further possible communications. On the other hand, the context where the publishing is made is not important and can be ignored. This is because, firstly, extensions to service compositions are made via invocations. Therefore, the composition's session is the session where the invocation is made. Secondly, internal session structures on the server side are usually hidden from clients. If the published service needs to communicate with an internal service in the server side, this can be done by further invocations or extra-session communications (as explained below).

In conclusion, if the invocation or the published services are working under an active session then the result can be one of the scenarios presented in Figure 5.1.

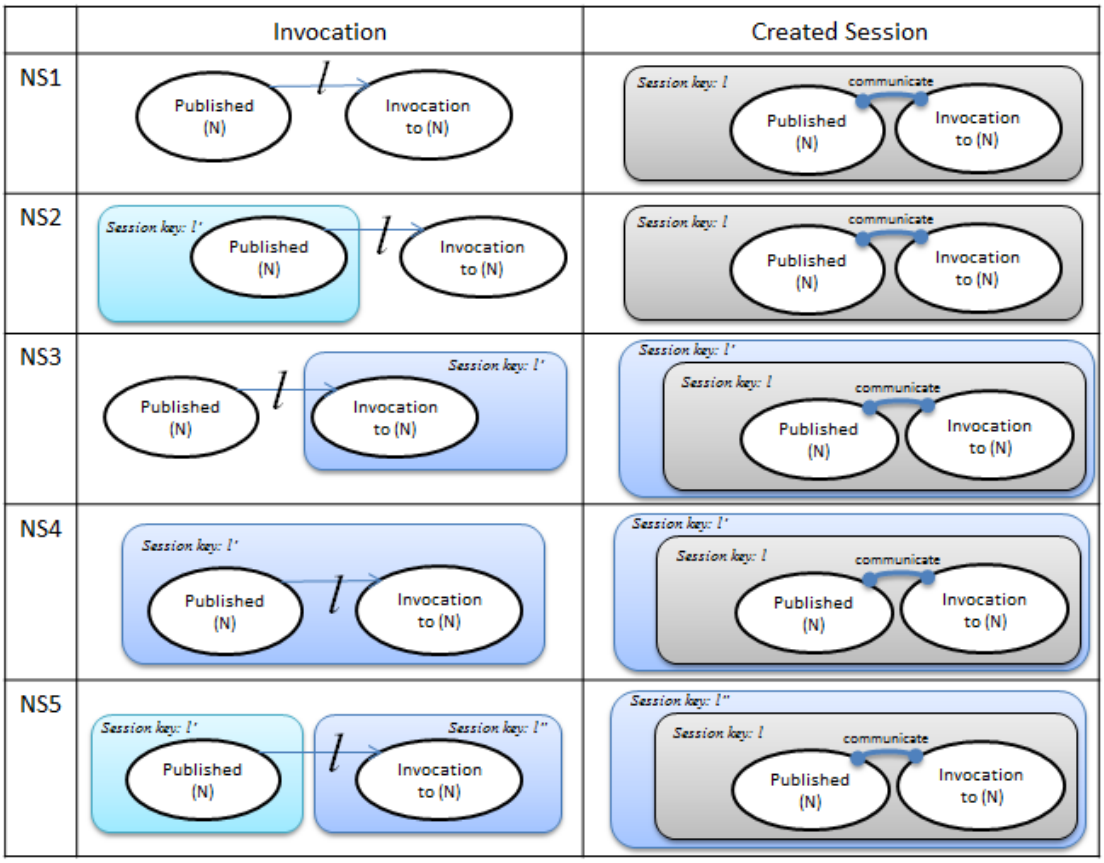


Figure 5.1: Scenarios for creating new sessions

The previous scenarios assume that a new session will always be created. In other words, invoked services are ready to join any new session offered by the published service as a new nested session of their current session (where the invocation is made) if this exists. Instead, the invocation service can request that the published service join its current session by using the idiom $N \Leftarrow^{+} \{q\}$, namely *join invocation*. In this case, the published service will be fully integrated into the current session and no new labels will be issued and no new session will be created. However, if designers decide to use join invocation, they should be aware that the published service will be able to communicate with any service in the session, not only the invocation service.

Considering the above discussion, if the join invocation service is used, then one

of the following scenarios applies:

- *The join invocation and the published service are not working under any session.* In this case, the join invocation will act as a normal invocation, where the published service creates a new label, then the published service passes this label to the invocation service to create a new session; see Figure 5.2.JS1.
- *The published service is under an active session.* In this case, the join invocation will act as a normal invocation, where the published service ignores the active session label and creates a new label, then the published service passes this label to the join invocation service to create a new session; see Figure 5.2.JS2.
- *The join invocation service is under an active session.* The new label created by the published service is ignored, and both the join invocation and the published service will join the running session (invocation service's session); see Figure 5.2.JS3.
- *Both the join invocation and the published service are working under the same active session.* The new label created by the published service is ignored, and both the join invocation and the published service will join the running session (invocation service's session); see Figure 5.2.JS4.
- *The join invocation and the published service are working under different active sessions.* Both the new label created by the published service and the label of the published service's active session are ignored, and both the join invocation and the published services will join the running session (invocation service's session); see Figure 5.2.JS5.

### 5.2.2 Persistent Services

The previous discussions assume that the services are only available once. In other words, if they are not recursively defined then they eventually terminate and disappear from the system. However, in SOC, published services are often considered as persistent, that is, they should remain in the system after serving one client. Moreover, they should be able to serve more than one client at a time.

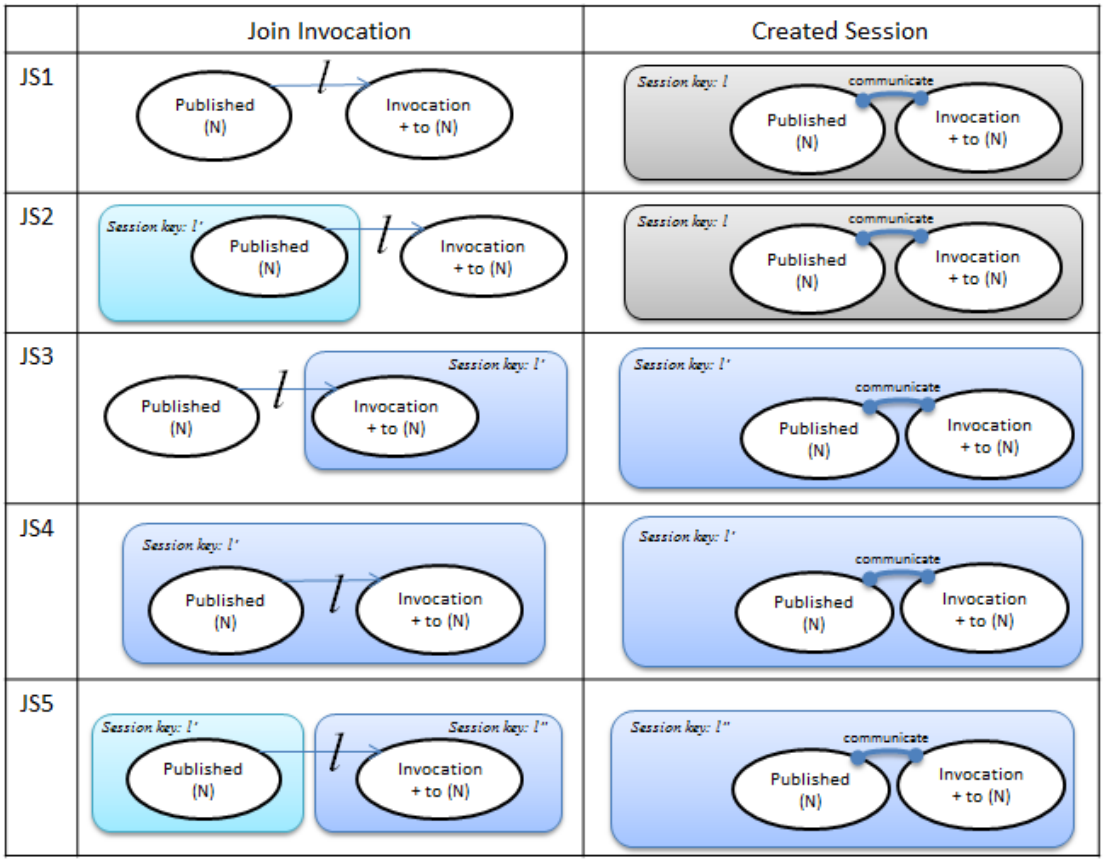


Figure 5.2: Scenarios for joining sessions

In CSPs, we add a replication operator ($*$) to work solely with published services to turn them to persistent published services. The replication operator and persistent published services are written as follows:

**Definition 5.2.** Let $p$ be a process. The process $*p$ represents an infinite number of copies of $p$ working in parallel: $*p = p \| p \| p \| \ldots$ (recall that $\|$, in CSPs, means that the copies do not synchronise on any event).

**Definition 5.3.** A persistent published service is written as $*(N \Rightarrow p)$, and can be abbreviated as $*N$. This indicates that an infinite number of copies of the service $N$ are available, so this service will persist in the system regardless how many times it has been invoked.

*Remark.* According to the definition of the replication operator, $*N = N \| N \| \ldots$,

which will be reduced to $N \Rightarrow p \parallel N \Rightarrow p \parallel \ldots$ and then later to $l_0 : p \parallel l_1 : p \parallel \ldots$.

This form of recursion does not violate the condition of guarded recursion stated in CSP (for guarded recursion in CSP refer to [84]), because each new copy of the published service is guarded with a new label.

Therefore, different copies in the replicated process do not synchronise on any event. In SOC paradigm, a persistent service delivers the same service to different clients in different sessions, which means, in essence, that these copies do not need to communicate. Formally speaking, the replication operator works with published services only. Therefore, the different copies of a persistent service will be labelled with different labels which means that these copies do not communicate. If communication is needed between these copies (e.g. sending shared tokens), then they can use unsessioned communications as explained below.

### 5.2.3 Communication between sessions

The bodies of the published and invocation services can be written as regular CSP processes where (as explained in Chapter 2) communication between processes can be carried on by the input/output primitives $(a?x, a!v)$. For instance, let $p = a?x \rightarrow SKIP$ be a process that inputs a value in $a$ then terminates successfully, and $q = a!4 \rightarrow SKIP$, a process that sends the value 4 via channel $a$ then terminates successfully. According to our sessioning algorithm the final result of $N \Rightarrow p \parallel_{\{a\}} N \Leftarrow \{q\}$ will be $l : p \parallel_{l:\{a\}} l : q$ which will be reduced to: $l : (a?x \rightarrow SKIP) \parallel_{l:\{a\}} l : (a!4 \rightarrow SKIP)$, where $l.a?x$ synchronises with $l.a!4$.

However, in some cases services need to communicate outside the boundaries of a session. For instance, in order to exchange information or return results to the upper session (if it exists), or to update a shared variable not related to any session.

In the first case, where services in a session need to communicate with services in the upper level, we use the primitives $(a \uparrow?x, a \uparrow!v)$. If the symbol $\uparrow$ is attached to a channel then this channel will be labelled with the upper session label. For instance, if an event $a?x$ is reduced to $l1.l2.a?x$ by the sessioning algorithm, then $a \uparrow?x$ will be reduced to $l1.a?x$, which allows this channel to communicate with the channel $a$ in the upper session. In a case in which the event works under no upper level, then the symbol $\uparrow$ removes the label if any. For instance, if an event $a?x$ is reduced to $l1.a?x$

by the sessioning algorithm, then $a \uparrow ?x$ will be reduced to $a?x$.

In the second case, where services in a session need to communicate with other processes in an unsessioned mode, we use the primitives $(a\diamondsuit?x, a\diamondsuit!v)$. If the symbol $\diamondsuit$ is attached to a channel then this channel will not be labelled with any session label. For instance, if an event $a?x$ is reduced, by the sessioning algorithm, to $l1.l2.a?x$ then $a\diamondsuit?x$ will be reduced to $a?x$, which allows this channel to communicate with the channels $a$ located outside the boundaries of any session.

### 5.2.4 Termination

If services are not defined recursively then they will eventually terminate. Ensuring graceful termination is very important in SOC. This is to avoid dangling processes, waiting for data from other processes in the system. Graceful termination means that when one side of a session terminates, it should inform the other participants, and the other participants should be able to respond and close or continue their executions.

In CSP, *SKIP* is used as an explicit termination process. It is used by processes to announce their successful termination. Formally speaking, *SKIP* is the process which evaluates the successful terminal event $(\surd)$ then terminates.

In CSPs, where the communications are structured in sessions, *SKIP* alone will not ensure graceful termination. The announcement of *SKIP* should be kept within the boundaries of sessions to avoid mixing announcements from different sessions. Therefore, in CSPs, when one participant in a session terminates, it evolves to a labelled *SKIP*, i.e *l:SKIP*, indicating that this participant in the session with label $l$ has terminated.

Similar to the notion of *synchronised termination* in CSP (see Chapter 3 page 49), in CSPs we allow the parallel composition to terminate when all participants are ready to terminate. However, in CSPs, processes might work in different layers due to sessions. Therefore, we introduce a levelling up mechanism by which a successfully terminated session informs its parent by changing the label of *SKIP* from its own label to its parent session label, i.e. level up the label; until there is no parent then *SKIP* is evaluated.

In other words, *SKIP* terminations in one session are synchronised, but *SKIP* terminations in different sessions are interleaved and their session labels are levelled

up. The levelling up is used to announce terminations to session in the higher level. The flow transition diagram in Figure 5.3 shows the propagation of *SKIP*.

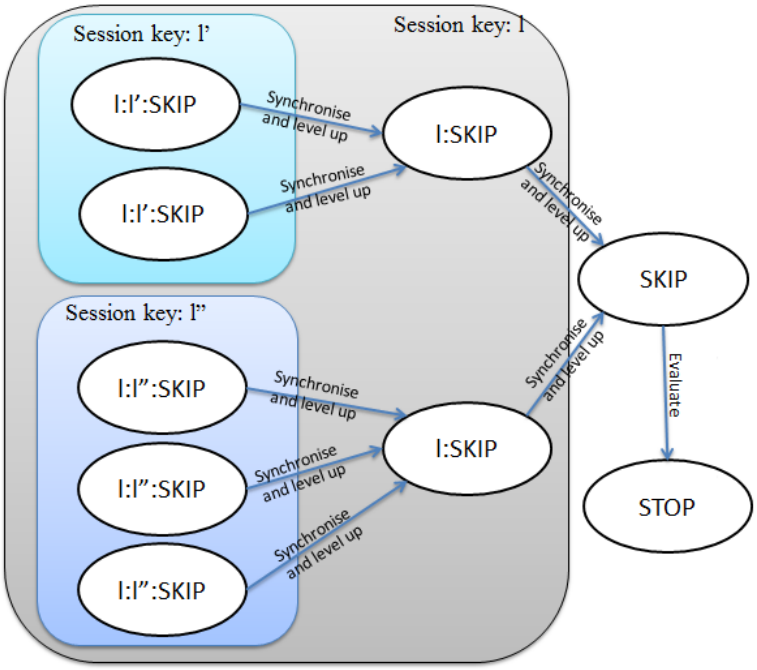


Figure 5.3: The propagation of *SKIP* termination process

Nested sessions in our model are not forced to terminate if the parent session terminates. This is because *SKIP* represent a successful termination, which means that the system is working as designed, so there is no need to interrupt the work of nested sessions.

The propagation of termination processes ensures the right order of announcements of terminations to other participants, but to allow these participants to have the option to respond and close or continue their executions, we add to CSPs the *LISTEN* primitive process.

When one of the session participants terminates then its siblings can interrupt their executions by using *LISTEN*. *LISTEN* can be understood as a special yielding process which responds to *SKIP* if there is any, or lets the process continue its execution, as illustrated by the following example:

**Example 5.1.** Let $p = a \rightarrow b \rightarrow SKIP$, $q = a \rightarrow b \rightarrow SKIP; LISTEN; c \rightarrow$

$d \to SKIP$, and $system = (N \Rightarrow p) \parallel_{\{a,b\}} (N \Leftarrow \{q\})$. According to our operational semantics (see Section 5.3.2), this will be reduced to: $l : (a \to b \to SKIP) \parallel_{\{l.a,l.b\}} l : (a \to b \to SKIP; LISTEN; c \to d \to SKIP)$

After evaluating $a$ and $b$ under the created session $l$, process $p$ will terminate its execution and announce it in the session. The *l:SKIP* announcement will allow the process $q$ to interrupt its execution because of the *LISTEN* construct, and terminate without executing $c$ and $d$.

*Remark.* Recall that the terminal event $\surd$ in *SKIP* will be hidden in the sequential composition (see the sequential composition operational semantics in Figure 2.2 page 42). Therefore, the first *SKIP* in $q$ will not evolve to *l:SKIP* but will evolve to the next process in the sequential composition of $q$ which is *LISTEN*.

If during the evaluation of the process *LISTEN* there was no announcement for *SKIP*, then *LISTEN* will behave like normal *SKIP*.

With labelled *SKIP* and *LISTEN*, the termination mechanism of CSPs ensures safe termination and additionally cleans the system from session labels generated during service invocations.

Optionally, the execution of *LISTEN* can trigger the execution of a predefined terminal handler by using the new operator $\blacktriangleright$. More precisely, if $p$ and $q$ are processes then $p \blacktriangleright q$ will behave as $p$ until it terminates with *LISTEN*, then the execution is passed to $q$.

*Remark.* We remark that diagrams in this section are illustrating diagrams to explain ideas, and they do not follow any known drawing standards like UML4SOA [11].

The previous discussion gave an informal overview of our model, highlighting the main features of the model. In the following section we describe the formal semantics of CSPs.

## 5.3 CSPs Semantics

In this section we develop the formal semantics of CSPs. We first present the syntax of CSPs, then its operational semantics.

### 5.3.1 CSPs Syntax

The syntax of CSPs is given in Figure 5.4. CSPs extends CSP's syntax (See Section 2.4.3) with operators to facilitate service definition, service invocation, sessions, and termination.

The syntax is divided into *design syntax*, which can be used by the designers in their models, and *runtime syntax*, which is used in the semantics only.

The **design syntax** includes the following:

*Definitions:* In addition to CSP process definitions ($N = p$; where $N$ is the name of the process $p$), published (invocable) services can be defined in CSPs using the definition idiom ($\Rightarrow$). In ($N \Rightarrow p$), $N$ is the service name and $p$ is its body. Starred service names ($^*N$) or starred published services ($*(N \Rightarrow p)$) represent persistent services.

*Processes:* In addition to CSP constructs, CSPs includes invocation services which can be defined using the invocation idiom ($\Leftarrow$) or the join operator ($\Leftarrow^+$). In ($N \Leftarrow \{q\}$) the published service $N$ is invoked by the client protocol $q$ ($q$ is a CSPs process).

In CSPs, services can terminate normally by *SKIP*, as in the original CSP, or they can interrupt their execution if their siblings in the same session have terminated. This interruption can be formally implemented by introducing the new terminal signal (event) † ,which represents listening to session closure. † can be installed in a process by using the termination process *LISTEN*, which listens to session closure and terminates.

In CSPs, synchronised terminal events announce the termination of a session. In the case in which a parent session terminates, its nested sessions are preserved until they internally terminate and they are not forced to terminate.

Moreover, a new operator $\blacktriangleright$ has been added to activate termination handlers in case a process has been terminated by *LISTEN*. In $p \blacktriangleright q$, $q$ is the termination handler of $p$, and $q$ will be evaluated if $p$ performs *LISTEN*.

Accordingly, we define $\Omega$ as the terminal events set, which contains $\{\surd, \dagger\}$. We use $\omega, \omega'$ to range over it. In the rest of the chapter, $\Sigma^\Omega$ denotes the set of the observable events in addition to the set of terminal events ($\Sigma^\Omega = \Sigma \cup \Omega$).

In CSPs, we allow observable events to be decorated with $\diamond, \uparrow$. If $a\diamond$ is used then the observable event $a$ will not get any labels. If $a \uparrow$ is used then the observable event

$a$ will get the label of the parent session which can be retrieved using the $f_p$ function. $f_p$ is defined later in the section.

The **runtime syntax** includes the following:

The *Label* operator is used to identify session boundaries. In the semantics, a process will be given a label during session creation, when an invocation and a published service work in parallel.

We use $l : p$ to denote a process $p$ which is working under session $l$, and $\underline{l} : p$ to denote a process $p$ which is working under a temporary label $\underline{l}$; a labelled process with a temporary label will not be reduced, until the label is replaced with a normal label. We assume $\mathcal{L}$ is an infinite set of session labels $l$, and $\underline{\mathcal{L}}$ is the set which contains $\underline{l}$ for each $l \in \mathcal{L}$. To generate a fresh label we use the function *new*.

In the model, sessions can be nested, therefore, $l : l' : p$ denotes a process that works under session $l'$ which is a nested session of session $l$, where $l, l' \in (\mathcal{L} \cup \underline{\mathcal{L}})$. We use $L : p$ where $L \in (\mathcal{L} \cup \underline{\mathcal{L}})^*$ to denote a chain of labels for process $p$, that is, $l_0 : ... : l_n : p \quad n \geq 0$. We use $[L :]p$ to denote a process $p$ which may or may not have a label.

Similarly, we use $l.a$ to denote an event $a$ which is evaluated under session $l$. We use $L.a$ to denote an event $a$ which is tagged with a chain of labels. Finally, $[L.]a$ denotes an event $a$ which may or may not have a label.

**Definition 5.4.** Let $L$ be the chain $l_0 : ... : l_n \quad n \geq 0$. The function $f_p : \mathcal{L}^* \rightarrow \mathcal{L}^*$ returns the parent session for the session $L$ by taking out the last added label, as follows:

- If $n = 0$ then $L$ is of the form $l_0$ and $f_p(l_0)$ returns empty label.
- If $n > 0$ then $L$ is of the form $l_0 : ... : l_n \quad n > 0$ and $f_p(L)$ returns $l_0 : ... : l_{n-1} \quad n \geq 0$.

*Structural congruence.* The structural congruence ($\equiv$) is the least congruence relation guarded by laws in Figure 5.5. These equalities are applied to labels $L \in \mathcal{L}^*$; not to underlined labels.

(Design Syntax)

| | | | |
|---|---|---|---|
| (Definitions) | $d ::=$ | $N = P$ | (Process definition) |
| | | $\mid N \Rightarrow P$ | (Service definition) |
| | | $\mid *(N \Rightarrow P)$ | (Persistent service) |
| (Processes) | $P, Q ::=$ | $\ldots$ | (CSP syntax) |
| | | $\mid *N$ | (Persistent service name) |
| | | $\mid N \Leftarrow \{P\}$ | (Service invocations) |
| | | $\mid N \Leftarrow^{+} \{P\}$ | (Join invocations) |
| | | $\mid P \blacktriangleright Q$ | (Termination handler) |
| | | $\mid N$ | (Service name) |
| | | $\mid LISTEN$ | (Primitive process) |

(Runtime Syntax)

$p, q ::= P$ (process) $\mid d$ (Definition) $\mid (new)(N \Rightarrow p)$ (New label) $\mid L : p$ (Label)
where $L \in (\mathcal{L} \cup \underline{\mathcal{L}})^*$

Figure 5.4: CSPs Syntax

$$
\begin{array}{cc}
L : (a \rightarrow p) \equiv L.a \rightarrow L : p & L : (a\diamond \rightarrow p) \equiv a \rightarrow L : p \\
L : (a \uparrow\!\rightarrow p) \equiv f_p(L).a \rightarrow L : p & L : (p \oplus q) \equiv L : p \oplus L : q \\
L : (p \backslash A) \equiv L : p \backslash L : A & L : (p[\![R]\!]) \equiv L : p[\![L : R]\!] \\
L : (p \|_A q) \equiv L : p \|_{L:A} L : q & L : (\mu p.f(p)) \equiv \mu p.f(L : p) \\
L : STOP \equiv STOP & L : (p \blacktriangleright q) \equiv L : p \blacktriangleright q
\end{array}
$$

where $L \in \mathcal{L}^*$, $\oplus \in \{\Box, \sqcap, |||, ;\}$, and if $a\, R\, b$ then $(L.a)\ L : R\ (L.b)$

Figure 5.5: CSPs Structural Congruence

### 5.3.2 Operational Semantics of CSPs

CSPs operational semantics extends the original CSP operational semantics that is presented in Section 2.4.3. To define the semantics of the new extensions in CSPs we need to update the event semantics and the labelled transition system of CSP first.

CSPs adopts the event semantics of CSP. However, because in CSPs service names and session labels play a significant role as explained in the previous section, we update this semantics as follows:

Firstly, we define $\mathbb{S}$ and $\mathbb{L}$ to be the sets of allocated service names and allocated labels respectively, where $\mathbb{L}$ includes single labels and all possible label chains.

Secondly, we define $\Lambda$ to be the set of service names, service definition events $(N\top, N\top l)$, service invocation events $(N\bot)$, and service join invocation events $(N+)$

as follows:

$\Lambda = \{N, N\top, N\bot, N+ \mid N \in \mathbb{S}\} \cup \{N\top l \mid N \in \mathbb{S} \wedge l \in \mathbb{L}\}$

where $N\top, N\top l, N\bot$, and $N+$ indicate the following:

- $N\top$ indicates that a process has been published as a service with name $N$.
- $N\top l$ indicates that the published service $N$ is part of the session with label $l$.
- $N\bot$ indicates that the published service $N$ is invoked by the current process.
- $N+$ indicates that the published service $N$ is asked to join the session of the invocation service.

We use $\lambda, \lambda'$ to range over $\Lambda$.

Finally, we update the universal set of observable events ($\Sigma$) to refer to ($\Sigma \cup \Lambda$), where the new $\Sigma$ denotes the set of process actions (the old $\Sigma$), service actions including invocations, definitions, and join invocations ($\Lambda$). We write $\Sigma^\tau$ to denote the new set of observable events with the silent event. $\Sigma^\Omega$ denotes the new set of observable events with the terminal events. $\Sigma^{\Omega\tau}$ denotes the set of observable events with the silent event and terminal events.

The LTS of CSPs is defined as a tuple $((CSPs, \mathcal{P}^{\mathcal{L}^*}), LTSlabels, \longrightarrow_{\mathcal{D}})$, where *CSPs* denotes the full space of CSPs processes, LTSlabels denotes the set of transition labels, $\mathcal{D}$ is a list of processes/services definitions, and $\longrightarrow_{\mathcal{D}} \subseteq (CSPs, \mathcal{P}^{\mathcal{L}^*}) \times LTSlabels \times (CSPs, \mathcal{P}^{\mathcal{L}^*})$. If $p, q \in CSPs$, $\mathbb{L} \in \mathcal{P}^{\mathcal{L}^*}$ and $a \in LTSlabels$, then the tuple $((p, \mathbb{L}), a, (q, \mathbb{L})) \in \longrightarrow_{\mathcal{D}}$ can be written as $(p, \mathbb{L}) \xrightarrow{a}_{\mathcal{D}} (q, \mathbb{L})$. The set $\mathbb{L}$ of allocated labels and the list $\mathcal{D}$ are global parameters of the system, so we will omit them in the rules and write just $p \xrightarrow{a} q$.

For the sake of simplicity, it will be assumed that configurations in the LTS are closed, with no free variables, specially that variables and parameters have no significant operational effects on our operators. Hence, no store will be defined. An extended operational semantics with store will be defined in Chapter 6 for the full calculus (soaCSP).

Below we present the operational semantics of CSPs. The rules explain the behaviour of the new extensions in CSPs whereas the operational semantics of the

original CSP operators are as presented in Section 2.4.3. However, to allow the previous rules of the operational semantics of the original CSP operators to evaluate labelled events as well as unlabelled events, we update $\Sigma$ to include labelled events as follows:

Firstly, we define $\mathbb{L} : \Sigma$ to be the set of all labelled events in the system, then we define the new $\Sigma$ to be $(\Sigma \cup \mathbb{L} : \Sigma)$. Similarly, we define $\mathbb{L} : \Omega$ to be the set of all labelled terminal events in the system, and we define the new $\Omega$ to be $(\Omega \cup \mathbb{L} : \Omega)$. Hence, $\Sigma^{\Omega}$ denotes the set of all unlabelled and labelled events in the system including unlabelled and labelled terminal events. We also define $\mathbb{L} : \Lambda$ to be the set of all labelled service events in the system, and we define the new $\Lambda$ to be $(\Lambda \cup \mathbb{L} : \Lambda)$.

**Process/service definitions:** If a process name appears in a script then this name should be replaced by its definition. We replace the name with only the body of the process if it is a regular process, using rule (def). We replace the name with the body of the process and keep the name if it is a service, using rule (def2). We replace the name with the body of the process and keep the name and the replication operator if it is a persistent service, using rule (ps-def).

$$(\text{def})\frac{}{N \xrightarrow{\tau} p}\, N = p \in \mathcal{D} \qquad (\text{def2})\frac{}{N \xrightarrow{\tau} N \Rightarrow p}\, N \Rightarrow p \in \mathcal{D}$$

$$(\text{ps-def})\frac{}{*N \xrightarrow{\tau} *(N \Rightarrow p)}\, N \Rightarrow p \in \mathcal{D}$$

**Label operator:** The structural congruence in Figure 5.5 allows us to use the CSP (prefix) rule (see Figure 2.2) for labelled observable events, and similarly for the other original operators of CSP.

The rules (labeli), (labelT), and (labelS) illustrate the behaviour of a labelled process when the original process evaluates silent action $\tau$, terminal events, or service events. The terminal and service events will be tagged with the label, but $\tau$ will not. Additionally, the rule (labelT) implements the levelling up process of the termination algorithm (explained in Section 5.2), by returning the same process with the label of

the upper level using the function ($f_p$).

$$(\text{labeli})\frac{p \xrightarrow{\tau} p'}{L : p \xrightarrow{\tau} L : p'} \quad (\text{labelT})\frac{p \xrightarrow{\omega} p'}{L : p \xrightarrow{L.\omega} f_p(L) : p}\ \omega \in \Omega$$

$$(\text{labelS})\frac{p \xrightarrow{\lambda} p'}{L : p \xrightarrow{L.\lambda} L : p'}\lambda \in \Lambda$$

In the following we define the operators which implement the sessioning algorithm:

**Published service:** The rules (srv-def1) and (srv-def2) implement the publishing process of the sessioning algorithm. The rule (srv-def1) allocates a new temporary label($\underline{l}$) via function ($new$) and the rule (srv-def2) gives this label to the service. This temporary label will be turned into a normal one when a session is created by rule (SE) (explained later in the section).

$$(\text{srv-def1})\frac{}{N \Rightarrow p \ \xrightarrow{N\top}\ (new)\,(N \Rightarrow p)}$$

$$(\text{srv-def2})\frac{}{(new)\,(N \Rightarrow p) \ \xrightarrow{N\top l}\ \underline{l} : p}\ l \notin \mathbb{L},\ \mathbb{L} := \mathbb{L} \cup \{l\}; \Sigma := \Sigma \cup \mathbb{L} : \Sigma$$

The rule (srv-def1) explains the behaviour of the function (new) which issues a fresh label, not in the set of the allocated labels $\mathbb{L}$, and rule (srv-def2) updates $\mathbb{L}$ with the new label.

**Persistent published service:** The rule (ps-wind) explains the behaviour of the replication operator ($*$). It shows that a persistent process will always have a new copy for processing, where all the copies work in parallel.

$$(\text{ps-wind})\frac{}{*(N \Rightarrow p) \ \xrightarrow{\tau}\ (N \Rightarrow p) \parallel *\,(N \Rightarrow p)}$$

**Invocation service:** The rules (srv-inv) and (joinS) implement the invocation process of the sessioning algorithm. The rules (srv-inv) and (joinS) show that invocations and join invocations respectively are ready to engage in any session offered by the published service $N$. Firstly, the process will be assigned a temporary label, then this

temporary label will be turned into a normal one when a session is created at rule (SE) (explained later in the section).

$$\text{(srv-inv)}\frac{}{N \Leftarrow \{q\} \xrightarrow{N\perp} \underline{l} : q}\, l \in \mathbb{L} \quad \text{(joinS)}\frac{}{N \Leftarrow^{+} \{q\} \xrightarrow{N+} \underline{l} : q}\, l \in \mathbb{L}$$

To complete implementing the sessioning algorithm we update the behaviour of the parallel composition operator to create a session whenever a published service and an invocation service for this published service are composed in parallel.

**Parallel composition operator:** The parallel composition operator is always provided with an interface set. This interface set contains the observable events which the designer wants the participants to synchronise on. The other events will be evaluated independently (interleaving mode).

The rules (par1), (par2), and (par3) in Figure 2.2 implement the normal behaviour of CSP parallel composition. The rule (par3) shows that the parallel processes synchronise on the events of the interface set of the parallel composition operator (i.e. $A$). The rules (par1) and (par2) show that processes will evaluate events outside the interface set independently.

Thanks to the new definition of $\Sigma$, the rules (par1,2) will work correctly in interleaving events outside the interface set regardless of whether these events are labelled/unlabelled observable events or labelled/unlabelled service events. The rule (par3) shows that parallel processes synchronise on all events of the interface set ($A$), and $A$ cannot contain service events or terminal events, for that we define below the behaviour of the parallel composition if these events need to synchronise in order to create or terminate sessions.

**Creating sessions** The process of creating a session, discussed in Section 5.2, is summarised in figures 5.1 and 5.2. If a published service works in parallel with an invocation service then a new session will be created and this session can be a nested session from the invocation side. If a published service works in parallel with a join invocation service then the published service will join the session of the invocation side; if there is none then it creates a new session. The formal definition of this process is as follows:

Firstly, we define the binary operation $([L.]\lambda, [L'.]\lambda') \rightarrow SL$ by the following table:

(SL-table)

| $[L.]\lambda$ | $[L.]N\top l$ | $[L.]N\top l$ | $[L.]N\top l$ | $[L.]N\top l$ |
|---|---|---|---|---|
| $[L'.]\lambda'$ | $N\bot$ | $L'.N\bot$ | $N+$ | $L'.N+$ |
| $SL$ | $l$ | $L' : l$ | $l$ | $L'$ |

where $(\lambda, \lambda' \in \Lambda)$, $(l, L, L' \in \mathbb{L})$, and the binary operation is commutative.

The (SL-table) is the formal representation of the invocations and the joint invocations' scenarios (see Section 5.2). In the table, $L$ and $L'$ can either be the same label chain or different chains.

Secondly we define the synchronisation rule (SE) where services synchronise and create a session. The decision as to whether the events $([L.]\lambda, [L.']\lambda' \in \Lambda)$ should synchronise and create a session or not relies on the existence of the tuple $([L.]\lambda, [L.']\lambda')$ in table (SL-table). Note that, $[\![SL/[L :]\underline{l}]\!]$ denotes the renaming of the label chain $[L :]\underline{l}$ with the label chain $SL$, and $SL : A$ denotes labelling of all events in the set $A$ with the label chain $SL$.

$$(\text{SE})\frac{[L :]p \xrightarrow{[L.]\lambda} [L :]\underline{l} : p' \quad [L' :]q \xrightarrow{[L'.]\lambda'} [L' :]\underline{l} : q'}{p \parallel_A q \xrightarrow{\tau} ([L :]\underline{l} : p')[\![SL/[L :]\underline{l}]\!] \parallel_{SL:A} ([L' :]\underline{l} : q')[\![SL/[L' :]\underline{l}]\!]}$$

where $([L.]\lambda, [L'.]\lambda') \in$ (SL-table).

**Terminating sessions** The process of terminating sessions was illustrated previously in Section 5.2. Terminal events are always synchronised in a parallel composition if they are in the same session as shown in rules (ParTses1) and (ParTses2) or without sessions as shown in rules (ParTS1), and (ParTS2). When terminal events are synchronised they terminate the session and announce this to the parent session by levelling up the label to the parent label using the function $f_p$. If all sessions in a session hierarchy are terminated then the parallel composition can terminate. The rules (ParTses1), (ParTses2), (ParTS1), and (ParTS2) replace (ParT1), (ParT2), and (ParT3) in Figure 2.2. On the other hand, terminal events are interleaved in a parallel composition if they are not at the same session as shown in rules (ParATses1) and (ParATses) below.

$$(\text{ParTses1})\frac{L : p \xrightarrow{L.\surd} f_p(L) : p \quad L : q \xrightarrow{L.\omega} f_p(L) : q}{L : p \parallel_A L : q \xrightarrow{L.\surd} f_p(L) : SKIP}\ \omega \in \{\surd, \dagger\}$$

$$(\text{ParTses2})\frac{L : p \xrightarrow{L.\dagger} f_p(L) : p \quad L : q \xrightarrow{L.\dagger} f_p(L) : q}{L : p \parallel_A L : q \xrightarrow{L.\dagger} f_p(L) : LISTEN}$$

$$(\text{ParTS1})\frac{p \xrightarrow{\surd} STOP \quad q \xrightarrow{\omega} STOP}{p \parallel_A q \xrightarrow{\surd} STOP}\ \omega \in \{\surd, \dagger\}$$

$$(\text{ParTS2})\frac{p \xrightarrow{\dagger} STOP \quad q \xrightarrow{\dagger} STOP}{p \parallel_A q \xrightarrow{\dagger} STOP}$$

$$(\text{ParATses1})\frac{L : p \xrightarrow{L.\omega} f_p(L) : p \quad [L' :]q \xrightarrow{[L'.]\omega'} q'}{L : p \parallel_A [L' :]q \xrightarrow{L.\omega} f_p(L) : p \parallel_A [L' :]q}\ \omega, \omega' \in \{\surd, \dagger\}$$

$$(\text{ParATses})\frac{[L :]p \xrightarrow{[L.]\omega} p' \quad L' : q \xrightarrow{L'.\omega'} f_p(L') : q}{[L :]p \parallel_A L' : q \xrightarrow{L'.\omega'} [L :]p \parallel_A f_p(L') : q}\ \omega, \omega' \in \{\surd, \dagger\}$$

The primitive process $LISTEN$ is defined as follows:

The rule (listen) shows that the process $LISTEN$ evaluates either of the terminal events † or $\surd$ then terminates.

$$(\text{listen})\frac{}{LISTEN \xrightarrow{\omega} STOP}\ \omega \in \{\surd, \dagger\}$$

**Termination handler:** The rules (T-hdr1), (T-hdr2), and (T-hdr3) implement the process of starting a termination handler in the case of a session termination.

$$(\text{T-hdr1})\frac{p \xrightarrow{a} p'}{p \blacktriangleright q \xrightarrow{a} p' \blacktriangleright q}\ (a \in \Sigma^{\tau})$$

$$(\text{T-hdr2})\frac{p \xrightarrow{\dagger} p'}{p \blacktriangleright q \xrightarrow{\tau} q} \qquad (\text{T-hdr3})\frac{p \xrightarrow{\surd} p'}{p \blacktriangleright q \xrightarrow{\surd} p'}$$

## 5.4 CSPs Properties

In this section, we investigate two properties of CSPs:

- We study the relationship between the session algorithm and the termination algorithm and show that the termination algorithm will gracefully terminate

any open sessions; if it is possible for it eventually to terminate. As mentioned in Section 5.2, sessions that include services which are defined recursively will never terminate, and will not be considered here.

- We study the effect of session labels on the behaviour of services. We show that sessions do not change the original behaviour of services. This is achieved by proving that the behaviour exhibited by a service under a session is equal to the behaviour exhibited by the same service under no session, if the same label is assigned to it afterwards.

Section 5.4.1 and Section 5.4.2 discuss the proofs of these two properties respectively. The rest of this section is devoted to defining the concepts and setting notations that will be used in following sections.

Let $\Sigma^*$ denote the set of all sequences produced from $\Sigma$; let $(\Sigma^*)^\Omega$ denote the set of all sequences which could possibly end with a terminal event from $\Omega$; $s, t, r, u$ are used to denote a single trace of a process; capital letters $S, T, R, U$ are used to denote a set of processes' traces. We use $\mathcal{T}$ to denote the set of all CSPs processes' traces.

**Definition 5.5.** A trace is a linear sequence of actions in $(\Sigma^*)^\Omega$ that a process can perform, excluding internal actions ($\tau$). A trace can be written as:

I. *Finite trace:* $\langle a_0, \ldots, a_n \rangle$, where $n \geqslant 0$ and $a_0, \ldots, a_{n-1} \in \Sigma \wedge a_n \in \Sigma^\Omega$, e.g. $\langle a, b \rangle$ where $a, b \in \Sigma$ denotes a trace of a process which will perform the observable event $a$ then the event $b$ with the possibility of evaluating internal actions before and after these events.

   As a particular case, we can have singleton traces and empty traces as follows:

   (a) *Empty trace:* $\langle\rangle$ which represents a process that can do nothing or a process which does a sequence of internal actions ($\tau$).

   (b) *Singleton trace:* $\langle a \rangle$, where $a \in \Sigma^\Omega$.

II. *Infinite trace:* $\langle a_0, \ldots, a_n, \ldots \rangle$, where $n \geqslant 0$ and $a_i \in \Sigma$ for all $i \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.

**Concatenation operator ($\frown$)** Let $s \in (\Sigma^*)^\Omega$ be a finite trace and let $t \in (\Sigma^*)^\Omega$ be a trace (finite or infinite) then $s \frown t$ denotes a trace containing the elements of $s$ followed by the elements of $t$, i.e. $s \frown t = \langle a_0, \ldots, a_n \rangle \frown \langle b_0, \ldots, b_n \rangle = \langle a_0, \ldots, a_n, b_0, \ldots, b_n \rangle$ or $s \frown t = \langle a_0, \ldots, a_n \rangle \frown \langle b_0, \ldots, b_n, \ldots \rangle = \langle a_0, \ldots, a_n, b_0, \ldots, b_n, \ldots \rangle$

**Label operator (:)** Let $s \in (\Sigma^*)^\Omega$ be a trace (finite or infinite) then $(L : s)$ is defined as follows, where $L \in \mathcal{L}^*$, $a \in \Sigma$, $\lambda \in \Lambda$, and $\omega \in \Omega$:

- $L : \langle\rangle = \langle\rangle$, because the empty trace has no elements.
- $L : (\langle a \rangle \frown s) = \langle L.a \rangle \frown L : s$
- $L : (\langle a \uparrow \rangle \frown s) = \langle fp(L).a \rangle \frown L : s$
- $L : (\langle a \diamond \rangle \frown s) = \langle a \rangle \frown L : s$
- $L : (\langle \lambda \rangle \frown s) = \langle L.\lambda \rangle \frown L : s$
- $L : \langle \omega \rangle = \langle L.\omega \rangle \frown fp(L) : \langle \omega \rangle$

Moreover, let $U \subseteq (\Sigma^*)^\Omega$ be a set of traces, then $L : U$ denotes labelling all the traces in the set $U$ with the label chain $L$, i.e. if $U = \{s, \ldots, t\}$ then $L : U = \{L : s, \ldots, L : t\}$.

### 5.4.1 Session termination

In this section, we prove that session labels do not affect the termination of services, and if services in a session terminate then the session will terminate as well. In other words, we ensure that session labels do not prevent services from terminating, and labels installed upon creation are removed in termination with respect to the hierarchy of sessions. This discussion will not develop a termination semantics of CSPs or discuss whether its processes will eventually terminate as been discussed in [84] and more recently in [74] for CSP (the foundations of CSPs); since such a discussion is outside of the scope of this thesis.

To facilitate this proof we firstly define terminated processes (adapted from Definition 3.4). In this section, processes and services are used interchangeability, because services in CSPs are processes.

**Definition 5.6** (Terminated Process)**.** Let $p$ be a CSPs process, then $p$ is a terminated process if it can not evolve more, and $p$ is terminating if there exists $p \xrightarrow{a_0} p_1 \xrightarrow{a_1} \ldots \xrightarrow{a_k} p_n \xrightarrow{a_n} p_{n+1} \nrightarrow$ where $n \geqslant 0$, and $\nrightarrow$ denotes that no further transitions are available.

If $a_n = \surd$ then this process successfully terminated, and if $a_n = \dagger$ then this process gracefully terminated, otherwise the process is blocked with no further transitions.

**Proposition 5.1.** *If a process $p$ is terminating then $L : p$ (i.e. a service under a session $L$) is terminating.*

*Proof.* In a sessioned process $L : p$, if $p$ terminates then according to Definition 5.6 there are three cases:

1. If $p$ successfully terminates, then the $\surd$ event is observable in the environment and the process will stop. According to rule (labelT) in Section 5.3.2 if a process evaluates the $\surd$ event then the sessioned process will evaluate the labelled $\surd$ event then evolve to $f_p(L)$ :*SKIP*, where $f_p(L)$ is the label of the parent session if exists. The evaluation of $f_p(L)$ :*SKIP* depends on the output of $f_p$ function. If there is a parent session (i.e. $f_p(L)$ :*SKIP* $\longrightarrow$ *L':SKIP*) then we have the following two cases:

   i. The resulting (*L':SKIP*) is the only *SKIP* in the parent session then the *SKIP* will use rule (labelT) in Section 5.3.2 to evaluate the termination event and announce the *SKIP* to the upper level using $f_p$ function again. The rules (ParATses1) and (ParATses) will ensure that this process will not be blocked in the parallel composition because of a terminated processes not in the same session.

   ii. The resulting (*L':SKIP*) synchronises with the other termination processes in the session using rule (parTses1) in Section 5.3.2 and announces the *SKIP* to the upper level using $f_p$ function.

   If there is no parent session (i.e. $f_p(L)$ :*SKIP* $\longrightarrow$ *SKIP*) then the *SKIP* process will be evaluated normally by using the rule (skip) in Figure 2.2.

2. If $p$ gracefully terminates, that is, $p$ evaluates *LISTEN*, then the $\dagger$ event is observable in the environment and the process will stop. According to rule (labelT)

in Section 5.3.2 if a process evaluates the † event then the sessioned process will evaluate the labelled † event then evolve to $f_p(L)$ :*LISTEN*, where $f_p(L)$ is the label of the parent session if exists. The evaluation of $f_p(L)$ :*LISTEN* depends on the output of $f_p$ function. If there is a parent session (i.e. $f_p(L)$ :*LISTEN* $\longrightarrow$ *L':LISTEN*) then we have the following three cases:

i. The resulting (*L':LISTEN*) is the only *LISTEN* in the parent session then *LISTEN* will use rule (labelT) in Section 5.3.2 to evaluate the termination event and announce the *LISTEN* to the upper level using $f_p$ function again. The rules (ParATses1) and (ParATses) will ensure that this process will not be blocked in the parallel composition because of a terminated processes not in the same session.

ii. The resulting (*L':LISTEN*) synchronises with (*L':SKIP*) in the session using rule (parTses1) in Section 5.3.2 and announces *SKIP* to the upper level using $f_p$ function.

iii. The resulting (*L':LISTEN*) synchronises with (*L':LISTEN*) in the session using rule (parTses2) in Section 5.3.2 and announces the *LISTEN* to the upper level using function $f_p$.

If there is no parent session (i.e. $f_p(L)$ :*LISTEN* $\longrightarrow$ *LISTEN*) then the *LISTEN* process will be evaluated normally by using rule (listen) in Section 5.3.2.

3. If $p$ is blocked, then the sessioned process will be blocked as well, because according to CSPs structural congruence in Figure 5.5, labels have no effect on the $STOP$ process.

□

### 5.4.2 The effects of labels on the behaviour of services

In this section, we show that sessions do not change the original behaviour of services. This is achieved by proving that the behaviour exhibited by a service under a session is equal to the behaviour exhibited by the same service under no session, if the same label is assigned to it afterwards.

For the purpose of this proof we define $opTr : CSPs \longrightarrow \mathcal{T}$ function, which works over our LTS using the relation $\overset{s}{\longmapsto} \subseteq CSPs \times (\Sigma^*)^\Omega \times CSPs$, where $s \in (\Sigma^*)^\Omega$ is a finite trace.

**Definition 5.7** (Trace derivation)**.** Let $s \in (\Sigma^*)^\Omega$ be a finite trace and $p, p' \in CSPs$ be CSPs processes, then the function $opTr : CSPs \longrightarrow \mathcal{T}$ is defined as:

$$opTr(p) = \{s | \exists p'.\ p \overset{s}{\longmapsto} p'\}$$

where $p \overset{s}{\longmapsto} p'$ is defined recursively as follows:

- $p \overset{\langle\rangle}{\longmapsto} p'$ iff $p = p' \vee \exists p_0, ..., p_n.\ p \overset{\tau}{\longrightarrow} p_0 \wedge ... \wedge p_n \overset{\tau}{\longrightarrow} p'$ where $n \geq 0$
- $p \overset{\langle a \rangle^\frown s}{\longmapsto} p'$ iff $\exists p'', p'''.\ p \overset{\langle\rangle}{\longmapsto} p''' \ \wedge\ p''' \overset{a}{\longrightarrow} p'' \ \wedge\ p'' \overset{s}{\longmapsto} p'$ where $a \in \Sigma^\Omega$, and elements of $\Omega$ can only occur as the last elements in $s$, that is, if $a \in \Omega$ then $s = \langle\rangle$ and $p' = STOP$.

**Theorem 5.1.** *For all $p \in CSPs$, $opTr(l : p) = l : opTr(p)$.*

*Proof.* We prove more generally that $L : opTr(p) = opTr(L : p)$.

By Definition 5.7 and the definition of traces label operator we have:

$L : opTr(p) = \{L : s | \exists p'.\, p \overset{s}{\longmapsto} p'\}$

We now need to prove:

$\{L : s | \exists p'.\, p \overset{s}{\longmapsto} p'\} = \{s' | \exists p''.\, L : p \overset{s'}{\longmapsto} p''\}$

To prove the above statement it is sufficient to prove that, for all $p \in CSPs$: $p \overset{s}{\longmapsto} p' \Leftrightarrow L : p \overset{s'}{\longmapsto} p''$, where $p'' = L : p'$, and $s' = L : s$.

In other words: $p \overset{s}{\longmapsto} p' \Leftrightarrow L : p \overset{L:s}{\longmapsto} L : p'$.

We prove first: $p \overset{s}{\longmapsto} p' \Rightarrow L : p \overset{L:s}{\longmapsto} L : p'$.

The proof proceeds by induction on the definition of the $opTr$ function.

By Definition 5.7 $p \overset{s}{\longmapsto} p'$ can result in one of two cases:

- $p \overset{\langle\rangle}{\longmapsto} p'$

  By definition 5.7: $p \overset{\langle\rangle}{\longmapsto} p'$ iff $p = p' \vee \exists p_0, ..., p_n . p \overset{\tau}{\longrightarrow} p_0 \wedge ... \wedge p_n \overset{\tau}{\longrightarrow} p'$ where $n \geq 0$.

  If $p = p'$ then $L : p = L : p'$ and according to the label operator $\langle\rangle = L : \langle\rangle$. Therefore, $L : p \overset{L:\langle\rangle}{\longmapsto} L : p'$

If $\exists p_0, ..., p_n.p \xrightarrow{\tau} p_0 \wedge ... \wedge p_n \xrightarrow{\tau} p'$ where $n \geq 0$ then according to the operational semantics rule (labeli) in Section 5.3.2:

(labeli): $p \xrightarrow{\tau} p' \Rightarrow L : p \xrightarrow{\tau} L : p'$.

Therefore, by definition 5.7: $L : p \overset{\langle\rangle}{\longmapsto} L : p'$.

Since $L : \langle\rangle = \langle\rangle$ by the definition of the traces label operator, then

$L : p \overset{L:\langle\rangle}{\longmapsto} L : p'$.

- $p \overset{\langle a\rangle^\frown s}{\longmapsto} p'$

  By definition 5.7: $p \overset{\langle a\rangle^\frown s}{\longmapsto} p'$ iff $\exists p'', p'''.p \overset{\langle\rangle}{\longmapsto} p''' \wedge p''' \xrightarrow{a} p'' \wedge p'' \overset{s}{\longmapsto} p'$ where $a \in \Sigma^\Omega$.

  By induction hypothesis: $L : p'' \overset{L:s}{\longmapsto} L : p'$, we also proved: $L : p \overset{L:\langle\rangle}{\longmapsto} L : p'''$. We now consider: $p''' \xrightarrow{a} p''$ only.

  When ($a \in \Sigma^\Omega$), $a$ can either be: an observable event, a service event, or a terminal event.

  1. If $a$ is an observable event, that is, $a, a\uparrow$, or $a\diamond$ then we have the following cases:

     i. If the observable event is $a$ then by the structural congruence we have: $p''' \xrightarrow{a} p''$ and therefore $L : p''' \xrightarrow{L.a} L : p''$.
     Thus, by definition 5.7, we have: $L : p \overset{\langle L.a\rangle^\frown L:s}{\longmapsto} L : p'$.
     Since $L : (\langle a\rangle^\frown s) = \langle L.a\rangle^\frown L : s$ by the definition of the traces label operator, we obtain $L : p \overset{L:(\langle a\rangle^\frown s)}{\longmapsto} L : p'$.

     ii. If the observable event is $a\uparrow$ then by the structural congruence we have: $p''' \xrightarrow{a\uparrow} p''$ and therefore $L : p''' \xrightarrow{f_p(L).a} l : p''$.
     Thus, by definition 5.7, we have: $L : p \overset{\langle f_p(L).a\rangle^\frown L:s}{\longmapsto} L : p'$.
     Since $L : (\langle a\uparrow\rangle^\frown s) = \langle f_p(L).a\rangle^\frown L : s$ by the definition of the traces label operator, we obtain $L : p \overset{L:(\langle a\uparrow\rangle^\frown s)}{\longmapsto} l : p'$.

     iii. If the observable event is $a\diamond$ then by the structural congruence we have: $p''' \xrightarrow{a\diamond} p''$ and therefore $L : p''' \xrightarrow{a} l : p''$.
     Thus, by definition 5.7, we have: $L : p \overset{\langle a\rangle^\frown L:s}{\longmapsto} L : p'$.

Since $L : (\langle a\diamond\rangle^\frown s) = \langle a\rangle^\frown L : s$ by the definition of the traces label operator, we obtain $L : p \overset{L:(\langle a\diamond\rangle^\frown s)}{\longmapsto} L : p'$.

2. If $a$ is a service event, i.e. $\lambda$, then the first step was obtained using (labelS) in Section 5.3.2, which implies that:
   $p''' \xrightarrow{\lambda} p''$ and therefore $L : p''' \xrightarrow{L.\lambda} L : p''$.
   Thus, by definition 5.7: $L : p \overset{\langle L.\lambda\rangle^\frown L:s}{\longmapsto} L : p'$.
   Since $L : (\langle\lambda\rangle^\frown s) = \langle L.\lambda\rangle^\frown L : s$ by the definition of the traces label operator, we obtain $L : p \overset{L:(\langle\lambda\rangle^\frown s)}{\longmapsto} L : p'$.

3. If $a$ is a terminal event, i.e. † or $\surd$, then the first step was obtained using (labelT) in Section 5.3.2, which implies that:
   $p''' \xrightarrow{\omega} p''$ and therefore $L : p''' \xrightarrow{L.\omega} f_p(L) : p'''$.
   To determine the transitions after $f_p(L) : p'''$ we should evaluate the $f_p$ function. According to Definition 5.4, the evaluation should be as follows:

   i. If $L = l_0 : ... : l_n$ and $n = 0$ then $f_p(l_0)$ returns empty label, i.e. $l_0 : p''' \xrightarrow{l_0.\omega} p''' \xrightarrow{\omega} STOP$.

   ii. If $L = l_0 : ... : l_n$ and $n > 0$ then $f_p(l_0 : ... : l_n)$ returns $l_0 : ... : l_{n-1}$, i.e. $l_0 : ... : l_n : p''' \xrightarrow{l_0...l_n.\omega} l_0 : ... : l_{n-1} : p''' \wedge l_0 : ... : l_{n-1} : p''' \xrightarrow{l_0...l_{n-1}.\omega} f_p(l_0 : ... : l_{n-1}) : p'''$.

   In step (ii), $f_p(l_0 : ... : l_{n-1})$ will be evaluated again, if $(n-1) = 0$ then the evaluation will stop by step (i), but if $(n-1) > 0$ then step (ii) will be repeated.

   According to Definition 5.4, $L'$ should be shorter than $L$ by one label. In addition, we know from Definition 5.7 that events in $\Omega$ are the last events in a process trace. Thus, $L$, in this case, can not include more labels as there are no more invocations. This shows that $f_p(L)$ will eventually reduced to empty, and $f_p(L) : p''' \overset{f_p(L):\langle\omega\rangle}{\longmapsto} STOP$.

   According to Definition 5.7, if $a$ is a terminal event then $s = \langle\rangle$ and $p' = STOP$.

   Thus, $STOP \overset{\langle\rangle}{\longmapsto} STOP$, and as been shown in Figure 5.5, labels have no effect on $STOP$ and $\langle\rangle = L : \langle\rangle$, so this implies:

$STOP \overset{L:\langle\rangle}{\longmapsto} L : STOP$.

By Definition 5.7:

$\exists p'', p'''. L : p \overset{L:\langle\rangle}{\longmapsto} L : p''' \ \wedge \ L : p''' \overset{L.\omega}{\longrightarrow} f_p(L) : p'' \ \wedge \ f_p(L) : p'' \overset{f_p(L):\langle\omega\rangle}{\longmapsto} STOP \ \wedge \ STOP \overset{L:\langle\rangle}{\longmapsto} L : STOP$

Since $L : (\langle\omega\rangle) = \langle L.\omega\rangle^\frown f_p(L) : \langle\omega\rangle$ by the definition of the traces label operator, we obtain $L : p \overset{L:\langle\omega\rangle}{\longmapsto} L : STOP$.

On the other direction, we reason similarly to prove $L : p \overset{L:s}{\longmapsto} L : p' \ \Rightarrow \ p \overset{s}{\longmapsto} p'$. That is, process $L : p$ can only execute $L : s$ and evolve to $L : p'$ where $s$ is obtained from $p$ performing $s$ and evolving to $p'$, which means, if $L : p \overset{s'}{\longmapsto} p''$ then $s' = L : s$, $p'' = L : p'$, and $p \overset{s}{\longmapsto} p'$.

We conclude that if $p \overset{s}{\longmapsto} p'$ then $L : p \overset{L:s}{\longmapsto} L : p'$, and if $L : p \overset{L:s}{\longmapsto} L : p'$ then $p \overset{s}{\longmapsto} p'$.

Therefore, we have proved that $\{L : s | \exists p'. \, p \overset{s}{\longmapsto} p'\} = \{s' | \exists p''. \, L : p \overset{s'}{\longmapsto} p''\}$, which implies that $L : opTr(p) = opTr(L : p)$, and this completes the proof.

□

## 5.5 Conclusions and Related Work

Multiparty sessions have been discussed in several papers that aim, like us, to develop new models to enhance the specification and verification techniques of multiparty sessions.

In this chapter, we have presented CSPs, where multiparty sessions are transparently created, manipulated, and terminated. Sessions are started with two parties and expanded by invocations. Designers have the option of integrating the new invoked service into the current session or creating a subsession, with the ability to communicate with the parent session.

This provides CSPs with a natural hierarchy of sessions, built upon service invocations only. This extends the elegant sessions binary tree hierarchy of the SCC calculi family [35, 93, 36], into a multi-edged tree hierarchy of sessions. In contrast, in the Conversation Calculus (CC) [137] and $\mu se$ [45] the sessions' structure spans through other structures like conversation endpoints and sites. In these calculi sessions can be

represented as the connection edges between different trees. Thus, the relationship between sessions and subsessions is different from CSPs trees. For instance, communicating with the parent session, which is available as extra session communication in CSPs, is available in CC by explicitly exchanging the conversation context (via *this*) between service parties then using this context in the extra communications.

In our session-based CSP, we adopt Hoare's concept of "labelling" to work in place of session keys. In CSPs, labels are used as fresh names which are attached dynamically to processes to construct sessions. This approach works in a similar way to the name-scoping mechanism *à la* $\pi$-calculus used in the SCC calculi family [35, 93, 36], to implicitly structure the interactions between services into a hierarchy of sessions. However, our *new* is a function not a binder, and sessions in the SCC family calculi are established between only two parties whereas in CSPs they are established between two or more parties.

Additionally, in SCC-like calculi SCC [35], SSCC [93], and CaSPiS [36] the invocation service is reduced to $r \triangleleft P$ and the published service is reduced to $r \triangleright P$ then they refresh the label at the parallel composition. However, it is not clear how they might agree on the $r$ label before the session is started. On the other hand, CSPs is clear on the order of creating labels, as explained in Section 5.2.

The calculi in [137, 45, 28, 87] support multiparty sessions, but they require session boundaries to be explicitly identified as conversation endpoints, intra-session names on sites, and session names in these calculi respectively.

CSPs invocation primitives allow dynamic expansion for sessions. This is also true in other calculi like [137, 45]. However, this not the case in [28, 87] as session participants are fixed in the design and expansions are not permitted.

In CSPs, deadlock-free sessions are not guaranteed by construction as in [28, 87], but deadlock-freedom can be asserted using the implementation of CSPs in FDR. Additionally, properties like protocol fidelity, which are usually proved by type checking [87, 136], could be achieved by developing a type system for CSPs. This is left for future work.

The calculi in [28, 87] capture the global scenarios of SOC systems following the choreography approach of WS-CDL [13]. On the other hand, CSPs is an orchestration calculus due to the orchestration primitives (invocations). However, the boundaries between the two design models are not firm, and orchestration calculi can model

service choreographies as presented in [47].

Additionally, in contrast to other orchestration calculi, processes can be defined in CSPs as ordinary processes or services. This allows CSPs to be extended as choreography calculus which captures the projection of a global view of the system, and we propose this as future work.

A significant feature of CSP is its explicit termination primitive, which reveals the termination state of a process to its environment, which suits services well. Therefore, in CSPs, we retain this feature and design our termination algorithm to gradually propagate successful terminations of services through the current hierarchy of sessions to ensure safe termination. This is done transparently without burdening designers with the overhead of maintaining session terminations. Moreover, we introduce a new primitive process which optionally allows processes to interrupt their execution and run a termination handler if their siblings have terminated.

In CSPs we preserve the context of subsessions until they terminate internally. In our view, this is reasonable as long as we assume that services will terminate successfully. Forcing subsessions to abort if parents terminate (like in [36]) is reasonable in those cases where parents fail to terminate normally.

The CSPs communication model is inspired by CSP. Thus, mixed synchronous and interleaving communications are inherited from CSP. However, in CSPs, we maintain these communications solely within sessions boundaries and avoid interference between sessions.

In CSPs, we limit the multicast communications of CSP within sessions. Therefore, new scenarios like (WS-coordination [16]) can be implemented easily in CSPs.

In CSPs, the extra-session communications include the out of session communications that interact with the upper context as the other SOC calculi. Additionally, because in CSPs we allow ordinary processes to be defined, we include non-session communications which allow interactions between ordinary processes and services. Thus, models which include an interaction between SOC systems and other systems like shared databases are easily encoded in CSPs.

CSPs does not admit sessions merge in two different hierarchies as in [45]; however, this will be achievable if mobile communication is permitted (see next chapter).

Session delegation as in [28] can be implemented in CSPs by invocation. However, a clearer scenario for delegation will be available if mobility is permitted in CSPs.

# 6

# CSP for Service-Oriented Architecture (soaCSP)

In this chapter we develop the soaCSP calculus as a formal modelling language for service-oriented computing systems. soaCSP is founded on the original CSP calculus. On top of the CSP modelling features, soaCSP defines asynchronous communications, mobile communications and session-based communications.

soaCSP comes as a result of combining the previous extensions: CSPa, $\mathcal{M}$CSP, and CSPs, which we developed in chapters 3, 4, and 5 respectively. In this chapter, we develop a communication model for soaCSP where the different kinds of communications of CSPa, $\mathcal{M}$CSP, and CSPs, can orchestrate to implement an expressive model for SOC systems.

We present the syntax of soaCSP, and the operational semantics of the calculus as we did for the previous extensions in this thesis.

Finally, we discuss the relationship between our calculus and the orchestration standard BPEL (see Chapter 2). We study the relationship by discussing the possibility of encoding BPEL constructs into soaCSP. We show that soaCSP is expressive enough for most BPEL functionality except constructs which deal with timed events, as soaCSP does not support time.

**Contributions of this chapter:**

1. Formally develop a communication model of soaCSP where the new kinds of communications (asynchronous communications, mobile communications and

session-based communications) can orchestrate.

2. In soaCSP, mobile channels can be sent synchronously and asynchronously, and data in a session can be sent synchronously and asynchronously.

3. In soaCSP, sessions merge can be achieved by transferring session labels along mobile channels.

4. Informally encode BPEL into soaCSP.

**Structure of this chapter:** Section 6.1 provides an informal description of the soaCSP model. Section 6.2 presents the full syntax and operational semantics of soaCSP. Section 6.3 discusses the relationship between soaCSP and the orchestration standard BPEL. Finally, Section 6.4 concludes the chapter and discusses the related works.

## 6.1 soaCSP Model

In Chapter 3 we developed a calculus for mixed synchronous, interleaving and asynchronous communications, namely CSPa. In CSPa we introduced implicit buffers to facilitate asynchronous communications on top of the synchronous/ interleaving communications of the original CSP. We use the $>, <$ symbols along with input/ output communications to indicate that this input consumes a value from the buffer/ outputs a value to a buffer.

In Chapter 4 we introduced mobility into CSP, by developing a new calculus, namely $\mathcal{M}$CSP. Our mobility model encodes the original $\pi$-calculus notion of mobility where channel names are sent through existing channels as a representation of transferring communication capabilities from one process to another. Additionally, our mobility model implements a new notion of mobility which changes the communication mode through a channel from synchronous to interleaving and vice versa.

In Chapter 5 we extended CSP with session-based communications. We call this new extension CSPs. In CSPs, we organise CSP communications into session hierarchies. The novelty in CSPs is that session hierarchies can be a mix of two-party sessions and multiparty sessions. An unlimited number of session hierarchies can be

composed in parallel where processes in a session can communicate with processes in the same session, with processes in the parent session, or with processes in the surrounding environment. In CSPs, if processes are defined as services then these processes can be a one copy service or an infinite number of copies persistent service.

In section 6.1.1 we discuss how the asynchronous extension will work with the mobility extension, so channels in soaCSP can be sent synchronously and asynchronously. Following that, in Section 6.1.2 we present how the session extension will work along with the result of combining the mobility extension and the asynchronous extension.

### 6.1.1 $\mathcal{M}$CSPa

In CSP, data is sent through channels either in a synchronous mode where data is received as soon as it has been sent, or in interleaving mode where data sending and receiving are performed independently. If processes' communications are executed in an interleaving mode then data will be lost. In CSPa, we manage to extend CSP with asynchronous communications, so data is sent to an implicit buffer (i.e. managed in the semantics), then when the receiving process are ready, it will consume the data from this buffer.

In $\mathcal{M}$CSP data can be channel names. Sending channel names in $\mathcal{M}$CSP affects the interface set of the parallel composition as explained in Section 4.2.

In this section, we discuss the consequences of storing the mobile channel names of $\mathcal{M}$CSP in the implicit buffers of CSPa, and we call the new model $\mathcal{M}$CSPa.

In $\mathcal{M}$CSPa, we allow channel names to be stored in the implicit buffers of other channels, but without their marks. This is because, marks are used to guide changes into the interface set of the parallel composition. Hence, in the case of asynchronous communications, carrying channels are not in the interface set of the parallel composition, and they can not update it without accessing it (i.e. be in the interface set).

More specifically, in $\mathcal{M}$CSPa, asynchronous communications can happen in the following case:

If $a!<(c)$ is executed then the mobile channel $c$ is stored in $a$'s buffer. Afterwards, $c$ can be retrieved by a process which performs $a?>(m)$.

### 6.1.2 Session-based $\mathcal{M}$CSPa (soaCSP)

In this section we discuss how the sessioning algorithm of CSPs affect the mobile communications of $\mathcal{M}$CSP, and the asynchronous communications of CSPa.

In the CSPa model (see Chapter 3 page 49) we assumed that the buffered-$\Sigma$ contains a buffer for all channels in the system. In the model of CSPs (see Chapter 5 page 119) we redefine the $\Sigma$ to include the labelled events. As a result, the buffered-$\Sigma$ in the model of soaCSP will have buffers for every labelled channel in the system. However, our model requires buffers to be created since the preprocessing step. In soaCSP, as in CSPs, the $\Sigma$ will be updated dynamically if a service is published. Therefore, we assume here if the $\Sigma$ is changed then new buffers will be created for the new channels, and these buffers will be added to the existing buffered-$\Sigma$.

We realise that the memory requirements of our model could be considered high, especially for complex big systems. Alternative solutions will be investigated in future work.

If channel names can be sent through labelled channels, which are within sessions, then we assume in soaCSP that mobile channel names will be labelled with session labels as well as carrying channels. Consider the following example:

**Example 6.1.** Let the following communication take place in session $l_1$:

$l_1.a!(l_1.c)^+ \longrightarrow l_1 : SKIP \parallel_{\{|l_1.a|\}} l_1.a?(m)^+ \longrightarrow l_1 : SKIP$

where $m$ is a variable mobile channel.

In this example, after the communication through channel $l_1.a$ is performed, the variable $m$ will have the value $l_1.c$, and the interface set will be updated with channel $l_1.c$.

As in the case of carrying channels, mobile channels can be tagged with $\uparrow, \diamond$ to indicate communications to the parent session and unsessioned communications respectively; for more details about $\uparrow, \diamond$ see Chapter 5 page 119.

This assumption could express useful scenarios like *temporary sessions merge*, see the following example:

**Example 6.2.** Let the following communication take place in a system, in two different sessions $l_1$ and $l_2$:

$l_1 : (a\diamond!(c)^+ \longrightarrow SKIP) \parallel_{\{|a|\}} l_2 : (a\diamond?(m)^+ \longrightarrow SKIP)$

where $m$ is a variable mobile channel.

After applying the label operator these communications will be as follows:

$a!(l_1.c)^+ \longrightarrow l_1 : SKIP \parallel_{\{|a|\}} a?(m)^+ \longrightarrow l_2 : SKIP$

Then, in this example, after the communication through channel $a$ is performed, the variable $m$ will have the value $l_1.c$, and the interface set will be updated with channel $l_1.c$. As a result, session $l_2$ can communicate with session $l_1$ if the connection $m$ is executed within session $l_2$; $m$ here can be considered as the key to session $l_1$ from session $l_2$.

In the following section we present soaCSP semantics where these new assumptions are formally defined.

## 6.2 soaCSP semantics

In this section we develop the formal semantics of soaCSP. We first present the syntax of soaCSP, then its operational semantics.

### 6.2.1 soaCSP Syntax

The syntax of soaCSP is given in Figure 6.1. soaCSP syntax is a combination of the CSPa, $\mathcal{M}$CSP, and CSPs syntaxes. Therefore, soaCSP extends CSP's syntax (See Chapter 2) with operators to facilitate asynchronous communications, mobile channel communications, service definition, service invocation, sessioning, and termination.

soaCSP syntax is divided into *design syntax*, which can be used by the designers in their models, and *runtime syntax*, which is used in the semantics only. Each of them is further divided into *events syntax* and *processes syntax*. *Design syntax* has also *definition syntax*.

**The events design syntax** Events in soaCSP, as in the standard CSP, can be classified as *simple events*, which are single literal names; *compound events*, which are literal names combined with other literal names or variables via the operator (.), e.g., $a$ or $a.b$; or *communication events*, which are simple or compound events composed with integer expressions to denote communicated data.

In addition to integers, soaCSP's channels can carry other channel names to facilitate mobility as been explained in Chapter 4 page 68. This is achieved by enclosing channel names in *brackets* and optionally decorating them with $(+, -, *)$ marks. The marks are used to allow processes to notify the parallel composition about the mode of communication on the communicated name.

Although the standard CSP allows data of different types to be communicated through channels, in soaCSP semantics, for simplicity, we assume that the communicated data are integers and literal names (channel names) only.

Channels of soaCSP can also input/ output data components into/ from implicit buffers, which are created in the semantics to facilitate asynchronous communications as been explained in Chapter 3 page 49. To access these buffers, channel names should be tagged with $>$ and $<$ symbols as follows:

- If the event $name?{>}INPC$ is used in a process then this process is ready to accept data from this channel's buffer (i.e. $B_a$ if the *name* is $a$).
- If the event $name!{<}OUTC$ is used within a process, then this process is ready to output data into this channel buffer (i.e. $B_a$ if the *name* is $a$).

In the syntax we use $INPC$ to denote input components and $OUTC$ to denote output components.

**The events runtime syntax** In addition to the design syntax events in runtime, the implicit buffers in soaCSP semantics can output values to processes and accept input from processes via the following events:

- If the event $name?{<}OUTC$ is used then the channel's buffer (i.e. $B_a$ if the name of the channel is $a$) is ready to accept data from the channel.
- If the event $name!{>}INPC$ is used then the channel's buffer is ready to send data into a process.

These events are reserved for buffers solely.

Additionally, in the semantics we use $l.a$ to denote an event $a$ which is evaluated under session $l$. We use $L.a$ to denote an event $a$ which is tagged with a chain of labels. Finally, $[L.]a$ denotes an event $a$ which may or may not have a label.

**The definitions design syntax** In addition to CSP process definitions ($N = p$; where $N$ is the name of the process $p$), published (invocable) services can be defined in soaCSP using the definition idiom ($\Rightarrow$).

In ($N \Rightarrow p$), $N$ is the service name and $p$ is its body. Starred service names ($^*N$) or starred published services ($*(N \Rightarrow p)$) represent persistent services.

**The processes design syntax** We extend the syntax of CSP's processes by defining invocation services as follows:

Invocation services are defined using the invocation idiom ($\Leftarrow$) or the join operator ($\Leftarrow^+$). In ($N \Leftarrow \{q\}$) the published service $N$ is invoked by the client protocol $q$ ($q$ is a soaCSP process).

In soaCSP, services can terminate normally by $SKIP$, as in the original CSP, or they can interrupt their execution if their siblings in the same session have terminated. This interruption can be formally implemented by introducing the new terminal signal (event) † ,which represents listening to session closure. † can be installed in a process by using the termination process $LISTEN$, where $LISTEN$ is the process that listens to session closure and terminates.

In soaCSP, synchronised termination is used to announce the termination of a session. In the case of a parent session terminating, its nested sessions are preserved until they internally terminate and they are not forced to terminate.

Additionally, we introduce a new operator $\blacktriangleright$ which can be used to activate termination handlers in case a process has been terminated by $LISTEN$. In $p \blacktriangleright q$, $q$ is the termination handler of $p$, and $q$ will be evaluated if $p$ performs $LSITEN$.

Accordingly, we define $\Omega$ as the terminal events set, which contains $\{\surd, \dagger\}$. We use $\omega, \omega'$ to range over it. In the rest of the chapter, $\Sigma^\Omega$ denotes the set of the observable events in addition to the set of terminal events set ($\Sigma^\Omega = \Sigma \cup \Omega$).

In soaCSPs, we allow observable events to be decorated with $\diamond, \uparrow$. If $a\diamond$ is used then the observable event $a$ will not get any labels. If $a \uparrow$ is used then the observable events $a$ will get the label of the parent session which can be retrieved using $f_p$ function. $f_p$ is defined later in this section.

**The processes runtime syntax** In the semantics, a process will be given a label during the creation of a session, i.e. when invocation and published services work in

parallel. Hence, the *Label* operator is used to identify session boundaries.

We use $l : p$ to denote a process $p$ which is working under session $l$, and $\underline{l} : p$ to denote a process $p$ which is working under a temporary label $\underline{l}$; a labelled process with a temporary label will not be reduced, until the label is replaced with a normal label. We assume $\mathcal{L}$ is an infinite set of session labels $l$, and $\underline{\mathcal{L}}$ is the set which contains $\underline{l}$ for each $l \in \mathcal{L}$. To generate a fresh label we use the function *new*.

In the model, sessions can be nested, therefore, $l : l' : p$ denotes a process that works under session $l'$ which is a nested session of session $l$, where $l, l' \in (\mathcal{L} \cup \underline{\mathcal{L}})$. We use $L : p$ where $L \in (\mathcal{L} \cup \underline{\mathcal{L}})^*$ to denote a chain of labels for process $p$, that is, $l_0 : ... : l_n : p \quad n \geq 0$. We use $[L :]p$ to denote a process $p$ which may or may not have a label.

**Definition 6.1.** Let $L$ be the chain $l_0 : ... : l_n \quad n \geq 0$ then the function $f_p : \mathcal{L}^* \to \mathcal{L}^*$ is defined as the function which returns the parent session for the session $L$ by taking out the last added label, as follows:

- If $n = 0$ then $L$ is of the form $l_0$ and $f_p(l_0)$ returns empty label.
- If $n > 0$ then $L$ is of the form $l_0 : ... : l_n \quad n > 0$ and $f_p(L)$ returns $l_0 : ... : l_{n-1} \quad n \geq 0$.

***Structural congruence.*** The structural congruence ($\equiv$) is the least congruence relation guarded by laws in Figure 6.2. These equalities are applied to labels $L \in \mathcal{L}^*$; not to underlined labels.

In Figure 6.2, we assume the following: $\oplus \in \{\Box, \sqcap, |||, ;\}$, and if $a \, R \, b$ then $(L.a) \; L : R \; (L.b)$. Moreover, $()^{\varphi}$ can be $()^+$, $()^-$, or $()$, and $\_$ can be ., ?, !, !<, ?>, !>, or ?<. We use $c$ to denote a mobile channel name, and $m$ to denote a variable mobile channel.

The structural congruence explains the behaviour of the label operator when it is applied to the different operators of soaCSP. Note that, when the label operator is applied to channels carrying other channels, then both channels will be labelled. However, if the carried data are integers data or variables, apart from their type, then this carried data will not be labelled.

***Design Syntax***

**[events]**

$EV ::= name?INPC$ (input) $\mid name!OUTC$ (output)
$\mid name?{>}INPC$ (asynchronous input) $\mid name.OUTC$ (compound)
$\mid name!{<}OUTC$ (asynchronous output) $\mid name$ (literal name)

(Input components)

$INPC ::= INP \mid INP?INPC \mid INP!OUTC \mid INP.OUTC$

(Input elements)

$INP ::= name$ (literal name) $\mid \ell$ (integer variable)
$\mid (name)$ (no change com) $\mid (name)^+$ (synchronous com)
$\mid (name)^-$ (interleaving com)

(Output components)

$OUTC ::= OUT \mid OUT?INPC \mid OUT!OUTC \mid OUT.OUTC$

(Output elements)

$OUT ::= name$ (literal name ) $\mid e$ (integer expression)
$\mid (name)$ (no change com) $\mid (name)^+$ (synchronous com)
$\mid (name)^-$ (interleaving com) $\mid (name)^{-*}$ (remove channel from $\alpha p$ )

(Definitions) $d ::= N = P$ (Process definition)
$\mid N \Rightarrow P$ (Service definition)
$\mid *(N \Rightarrow P)$ (Persistent service)

(Processes) $P, Q ::= \ldots$ (CSP syntax, see Figure 2.2)
$\mid *N$ (Persistent service name)
$\mid N \Leftarrow \{P\}$ (Service invocations)
$\mid N \Leftarrow^+ \{P\}$ (Join invocations)
$\mid P \blacktriangleright Q$ (Termination handler)
$\mid N$ (Service name)
$\mid LISTEN$ (Primitive process)

***Runtime Syntax***

**[events]**

$a ::= name?{<}OUTC$ (buffers input) $\mid L : a$ (Labelled a, $L \in (\mathcal{L} \cup \underline{\mathcal{L}})^*$)
$\mid name!{>}INPC$ (buffers output) $\mid EV$

**[processes]**

$p, q ::= L : p$ (Labelled p, $L \in (\mathcal{L} \cup \underline{\mathcal{L}})^*$) $\mid d$ (definition) $\mid P$ (process)
$\mid (new)(N \Rightarrow p)$ (new label) $\mid B_\Sigma$ (buffered$\Sigma$)

Figure 6.1: CSPs Syntax

$$
\begin{array}{ll}
L : (a \to p) \equiv L.a \to L : p & L : (a\diamond \to p) \equiv a \to L : p \\
L : (a \uparrow\!\to p) \equiv f_p(L).a \to L : p & L : (p \oplus q) \equiv L : p \oplus L : q \\
L : (a\_x \to p) \equiv L.a\_x \to L : p & L : (a \diamond \_x \to p) \equiv a\_x \to L : p \\
L : (a\_(m)^{\varphi} \to p) \equiv L.a\_(m)^{\varphi} \to L : p & L : (a \diamond \_(m)^{\varphi} \to p) \equiv a\_(m)^{\varphi} \to L : p \\
L : (a \uparrow \_x \to p) \equiv f_p(L).a\_x \to L : p & L : (a \uparrow \_(m)^{\varphi} \to p) \equiv f_p(L).a\_(m)^{\varphi} \to L : p \\
L : (m \to p) \equiv m \to L : p & L : (m\_.x \to p) \equiv m\_x \to L : p \\
L : (p\backslash A) \equiv L : p\backslash L : A & L : (p[\![R]\!]) \equiv L : p[\![L : R]\!]) \\
L : (p\|_A q) \equiv L : p\|_{L:A} L : q & L : (\mu p.f(p)) \equiv \mu p.f(L : p) \\
L : STOP \equiv STOP & L : (p \blacktriangleright q) \equiv L : p \blacktriangleright q \\
L : (a\_(c)^{\varphi} \to p) \equiv L.a\_(L.c)^{\varphi} \to L : p & L : (a\_(c\diamond)^{\varphi} \to p) \equiv L.a\_(c)^{\varphi} \to L : p \\
L : (a \diamond \_(c)^{\varphi} \to p) \equiv a\_(L.c)^{\varphi} \to L : p & L : (a \diamond \_(c\diamond)^{\varphi} \to p) \equiv a\_(c)^{\varphi} \to L : p
\end{array}
$$

$$
\begin{array}{rl}
L : (a\_(c\uparrow)^{\varphi} \to p) & \equiv L.a\_(f_p(L).c)^{\varphi} \to L : p \\
L : (a \diamond \_(c\uparrow)^{\varphi} \to p) & \equiv a\_(f_p(L).c)^{\varphi} \to L : p \\
L : (a \uparrow \_(c)^{\varphi} \to p) & \equiv f_p(L).a\_(L.c)^{\varphi} \to L : p \\
L : (a \uparrow \_(c\diamond)^{\varphi} \to p) & \equiv f_p(L).a\_(c)^{\varphi} \to L : p \\
L : (a \uparrow \_(c\uparrow)^{\varphi} \to p) & \equiv f_p(L).a\_(f_p(L).c)^{\varphi} \to L : p
\end{array}
$$

Figure 6.2: CSPs Structural Congruence

### 6.2.2 soaCSP Operational Semantics

soaCSP is an extension of the original CSP with SOC primitives which include asynchronous communications, mobile communications, and session-based communications, as well as defining service invocations and service publications. Therefore, in this section we present the new rules which implement these new capabilities on top of the original semantics of CSP which was presented in Figure 2.2 page 42.

The operational semantics of soaCSP is an integration of CSP operational semantics, CSPa operational semantics, $\mathcal{M}$CSP operational semantics, and CSPs operational semantics.

The integration process begins with defining the LTS of the full operational semantics. Following that, we introduce new rules to formally define the behaviour of the soaCSP model when these different extensions need to work together as been informally discussed in Section 6.1.

In soaCSP, the LTS is defined as a tuple $(((Proc, \sigma), \mathcal{P}^{\Sigma}, \mathcal{P}^{\mathcal{L}^*}), LTSlabels, \longrightarrow_{\mathcal{D}})$, where *Proc* denotes the full space of soaCSP processes, LTSlabels denotes the set of transition labels, $\mathcal{D}$ is a list of processes/services definitions, $\mathcal{P}^{\Sigma}$ is the set of parts of $\Sigma$, $\mathcal{P}^{\mathcal{L}^*}$ is the set of allocated labels, and $\longrightarrow_{\mathcal{D}} \subseteq ((Proc, \sigma), \mathcal{P}^{\Sigma}, \mathcal{P}^{\mathcal{L}^*}) \times$

$LTSlabels \times ((Proc, \sigma), \mathcal{P}^{\Sigma}, \mathcal{P}^{\mathcal{L}^*})$. If $p, q \in Proc$, $\mathbb{L} \in \mathcal{P}^{\mathcal{L}^*}$, $\alpha p, \alpha q \in \mathcal{P}^{\Sigma}$, and $a \in LTSlabels$, then the tuple $(((p, \sigma), \alpha p, \mathbb{L}), a, ((q, \sigma), \alpha q, \mathbb{L})) \in \longrightarrow_{\mathcal{D}}$ can be written as $((p, \sigma), \alpha p, \mathbb{L}) \xrightarrow{a}_{\mathcal{D}} ((q, \sigma), \alpha p, \mathbb{L})$. The set $\mathbb{L}$ and the list $\mathcal{D}$ are global parameters of the system, so we will omit them in the rules and write just $((p, \sigma), \alpha p) \xrightarrow{a} ((q, \sigma), \alpha q)$. For the sake of clarity, we also omit the second component of the LTS configurations, writing transitions simply as $(p, \sigma) \xrightarrow{a} (q, \sigma)$; we explicitly indicate with $(*)$ the rules that change the second component.

As can be seen in the LTS of soaCSP, the basic configuration (denoted by $C$) is a pair $(p, \sigma)$, where $\sigma$ is a local store. $\sigma$ is a collection of integer locations and literal name locations. We use $\sigma$, $\sigma'$,... to represent its different states. We write $\sigma(x)$ to denote the value of the variable $x$ in $\sigma$.

However, this form of configuration is not yet sufficient. Consider the case where a process $p$ is composed in parallel with $q$ (i.e. $(p\|_A q, \sigma)$). Assume $p$ performs an event $x$, then there is a transition from this initial configuration to a new configuration reflecting this change. Following the (par1) rule in Chapter 2 (updated with the store):

$$\frac{(p, \sigma) \xrightarrow{a} (p', \sigma')}{(p \ \|_A \ q, \sigma) \xrightarrow{a} NC}$$

where NC is a new configuration which has the process $p'$ in $\sigma'$ composed in parallel with $q$ which is still in $\sigma$. Since $\sigma'$ and $\sigma$ may differ, the composition operators (which were defined to work on processes) will be promoted to work on configurations. We use $\psi$ to denote the full space of configurations. The promoted versions of operators (where processes' stores might differ) are defined as follows:

$$\psi, \psi' \ ::= \ C \mid \psi \Box \psi' \mid \psi \|_A \psi' \mid \psi; \psi' \mid \psi \backslash a \mid \psi[\![R]\!] \mid \psi ||| \psi$$

Note that, the state of the local store of a process changes if mobile channels or data integers are communicated. Apart from this the store remains the same. In the semantics, $e$ is evaluated according to the standard integer semantics. Literal names can be any name, however for the sake of clarity *we use c to denote constant channel names, and we use m to denote variable mobile channel names.*

Before presenting the operational semantics of soaCSP, we summarise below the event types in soaCSP.

- **Observable events**, which are grouped in $\Sigma$ and can be simple names (e.g. $a$), compound names (e.g. $a.b$), or channels carrying integers, channel names, or a combination of them (e.g. $a.x$, where $x$ is an integer, or $a.(c)^{\varphi}$, where $c$ is a channel name).

  Note that, channels carrying data represent a class of events in $\Sigma$ rather than one single value, e.g., $c.Z$ represents the channel $c$ which can communicate any integer value ($c.Z$ means $\{c.x | x \in Z\} \subseteq \Sigma$). Inputting processes are able to receive any element of type $Z$, whereas outputting processes are only able to communicate one value [124]. Therefore, if $c.Z$ is an input event then it is actually an external choice over all the values of type integer, which is represented elegantly as $c?x$. If $c.Z$ is an output event then it is a single integer value, $c.x$ where $x \in Z$. Output events are represented as $c!x$. In our semantics, $c.x$ always equals $c!x$, except when $x$ is not in the store of the process which indicates that this event is a declaration for a new input variable.

  Similarly, channels carrying other channels represent a class of events in $\Sigma$.

  **Definition 6.2.** $c.CH$ denotes a channel $c$ that can communicate any channel name in $\Sigma$, that is

  $$c.CH = \{c.(b)^{\varphi} | b \in channels(\Sigma) \wedge \varphi \in \{\emptyset, +, -, -*\}\}$$

  where if $\varphi = \emptyset$, then the communication has no marks (i.e. $a.(c)$), and $channels(\Sigma)$ is a generalisation of the $channel(c.v)$ function [84], which extracts the channel name of an event; $channels(\Sigma)$ will return the set of all channel names of $\Sigma$'s events.

  Throughout what follows, if $c$ is a channel in $\Sigma$ then $\{|c|\}$ is the enumeration of $c$'s class of events in $\Sigma$.

  Channels can simultaneously carry multiple values in both directions. For instance, if $c$ is a channel name then the event $(c?x!y!z)$ is a valid event which accepts an integer and sends two integers.

  We range over the set $\Sigma$ by $a, b, \ldots$.

- **Service events**, which are grouped in $\Lambda$ and can be service definition events $(N\top, N\top l)$, service invocation events $(N\bot)$, or service join invocation events $(N+)$. $\Lambda$ set is defined as:

$$\Lambda = \{N, N\top, N\bot, N+ \mid N \in \mathbb{S}\} \cup \{N\top l \mid N \in \mathbb{S} \wedge l \in \mathbb{L}\}$$

  where $N, N\top, N\bot, N+, N\top l, \mathbb{S}, \mathbb{L}$ are as defined in Section 5.3.2.

  We range over the set $\Lambda$ by $\lambda, \lambda', \lambda_1, \ldots$.

- **Terminal events**, which are grouped in $\Omega$ and can be $\dagger$ or $\surd$.

  We range over the set $\Omega$ by $\omega, \omega', \omega_1, \ldots$.

- **Buffered events**, which are grouped in $\mathcal{B}$ and can be: in-buffer events $(a\rightarrow)$, or out-buffer events $(a\leftarrow)$. $\mathcal{B}$ set is defined as:

$$\mathcal{B} = \{a\rightarrow \mid a \in channels(\Sigma)\} \cup \{a\leftarrow \mid a \in channels(\Sigma)\}$$

  where $a\rightarrow, a\leftarrow$ are as defined in Section 3.2.

  We range over the set $\mathcal{B}$ by $a\rightarrow, b\rightarrow, \ldots$ and $a\leftarrow, b\leftarrow, \ldots$.

- **Silent event**, which is $\tau$.

Throughout what follows we write $\Sigma$ to denote the universal set of all observable events in the model, and we define $\mathbb{L} : \Sigma$ to be the set of all labelled observable events in the system, then we define the new $\Sigma$ to be $(\Sigma \cup \mathbb{L} : \Sigma)$. Similarly, we define $\mathbb{L} : \Omega$ to be the set of all labelled terminal events in the system, and we define the new $\Omega$ to be $(\Omega \cup \mathbb{L} : \Omega)$; we write $\Sigma^{\Omega}$ to denote the set of labelled and unlabelled observable events with the labelled and unlabelled terminal events. In addition, we define $\mathbb{L} : \Lambda$ to be the set of all labelled service events in the system, and we define the new $\Lambda$ to be $(\Lambda \cup \mathbb{L} : \Lambda)$; $\Sigma^{\Lambda}$ denotes the set of labelled and unlabelled observable events with the labelled and unlabelled service events. In the same way, we define $\mathbb{L} : \mathcal{B}$ to be the set of all labelled buffered events in the system, and we define the new $\mathcal{B}$ to be $(\mathcal{B} \cup \mathbb{L} : \mathcal{B})$; we use $\Sigma^{\mathcal{B}}$ to denote the set of labelled and unlabelled observable events with the labelled and unlabelled service events. We also write $\Sigma^{\tau}$ to denote the set of labelled and unlabelled observable events with the silent event. Any combination of

the above sets is written with one $\Sigma$, e.g., $\Sigma^{\Omega\tau}$ is used to denote the set of observable events with the silent event and terminal events, i.e. $\Sigma^{\Omega} \cup \Sigma^{\tau}$. Similarly, $\Sigma^{\Lambda\Omega\tau}$ is used to denote the set of observable events with service events, terminal events, and the silent event, i.e. $\Sigma^{\Lambda} \cup \Sigma^{\Omega} \cup \Sigma^{\tau}$. Finally, we write $\Sigma^{\Lambda\mathcal{B}\Omega\tau}$ to denote the set of all events in the model whether labelled or not.

Hereafter, we present the soaCSP operational semantics. Note that, in the semantics, $e$ is evaluated according to the standard integer semantics.

Firstly, Figure 6.3 presents the part of soaCSP semantics which includes the updated semantics of CSP with the store; the original semantics of CSP was presented in Figure 2.2 page 42. In addition to the store we update the rules to work with the new types of events as follows:

In the *External choice* operator, we allow any event type to resolve the choice, except the silent event, as in the standard CSP. The *Hiding* operator allows observable and buffered events to be hidden; however, we note that if a buffer event is hidden and the buffered-$\Sigma$ is not in the scope of the operator, then the connection with the buffered-$\Sigma$ is lost and the data will not be delivered/ retrieved from the buffered-$\Sigma$. Terminal events and service events cannot be hidden. In the *Renaming* operator, we only allow observable events and buffered events to be renamed, on the condition that the renaming relation is defined between events from the same type, that is, an observable event to an observable event and a buffered event to a buffered event. The Interleaving, the Sequential, and the Internal choice operators behave similarly to their original version in Figure 2.2 page 42.

Secondly, Figure 6.4 presents the updated operational semantics of the CSP *prefix operator*. This includes the channel semantics of $\mathcal{M}$CSP, and the rules of asynchronous communications and buffer communications of CSPa updated with stores. The explanations of these rules are presented in chapters 4 page 68 and Chapter 3 page 49 respectively.

In the Figure 6.4 the new rules (asy-out2), (asy-in2), (buffer-in2), and (buffer-out2), implement the behaviour of $\mathcal{M}$CSPa which has been presented informally in Section 6.1. They show that if a mobile channel is communicated asynchronously then the channel name will be stored in the buffered-$\Sigma$ and will be available to be retrieved by a process that is accepting data from the buffered-$\Sigma$ in the same way as has been explained for normal data in Chapter 3 page 49.

$$(\text{skip})\frac{}{(SKIP,\sigma)\xrightarrow{\surd}(STOP,\sigma)} \quad (\text{exch1,2})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\,\Box\,q,\sigma)\xrightarrow{a}(p',\sigma')} \quad \frac{(q,\sigma)\xrightarrow{a}(q',\sigma')}{(p\,\Box\,q,\sigma)\xrightarrow{a}(q',\sigma')} \quad (a\in\Sigma^{\Lambda\mathcal{B}\Omega})$$

$$(\text{exch3,4})\frac{(p,\sigma)\xrightarrow{\tau}(p',\sigma)}{(p\,\Box\,q,\sigma)\xrightarrow{\tau}(p'\,\Box\,q,\sigma)} \quad \frac{(q,\sigma)\xrightarrow{\tau}(q',\sigma)}{(p\,\Box\,q,\sigma)\xrightarrow{\tau}(p\,\Box\,q',\sigma)} \quad (\text{rec})\frac{}{(\mu p.f(p),\sigma)\xrightarrow{\tau}(f[\mu p.f(p)/p],\sigma)}$$

$$(\text{inch1,2})\frac{}{(p\sqcap q,\sigma)\xrightarrow{\tau}(p,\sigma)} \quad \frac{}{(p\sqcap q,\sigma)\xrightarrow{\tau}(q,\sigma)}$$

$$(\text{hid1})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\setminus A,\sigma)\xrightarrow{a}(p'\setminus A,\sigma')}\,(a\in\Sigma^{\Lambda\mathcal{B}\tau}\ \wedge\ a\notin A)$$

$$(\text{hid2,3})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\setminus A,\sigma)\xrightarrow{\tau}(p'\setminus A,\sigma')}\,(a\in\Sigma^{\mathcal{B}}\ \wedge\ a\in A) \quad \frac{(p,\sigma)\xrightarrow{\omega}(p',\sigma)}{p\setminus A\xrightarrow{\omega}(p',\sigma)}\,(\omega\in\Omega)$$

$$(\text{rem1,2})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p[\![R]\!],\sigma)\xrightarrow{b}(p'[\![R]\!],\sigma')}\,(a\in\Sigma^{\mathcal{B}}\ \wedge\ a\,R\,b) \quad \frac{(p,\sigma)\xrightarrow{a}(p',\sigma)}{(p[\![R]\!],\sigma)\xrightarrow{a}(p'[\![R]\!],\sigma)}\,(a\in\Lambda\cup\{\tau\})$$

$$(\text{rem3})\frac{(p,\sigma)\xrightarrow{\omega}(p',\sigma)}{(p[\![R]\!],\sigma)\xrightarrow{\omega}(p',\sigma)}\,(\omega\in\Omega)$$

$$(\text{seq1,2})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\,;q,\sigma)\xrightarrow{a}(p',\sigma')\,;(q,\sigma)}\,\left(a\in(\Sigma^{\Lambda\mathcal{B}\Omega\tau}-\{\surd\})\right)\ \frac{(p,\sigma)\xrightarrow{\surd}(p',\sigma)}{(p\,;q,\sigma)\xrightarrow{\tau}(q,\sigma)}$$

$$(\text{intv1,2})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\,|||\,q,\sigma)\xrightarrow{a}(p',\sigma'),|||\,(q,\sigma)} \quad \frac{(q,\sigma)\xrightarrow{a}(q',\sigma')}{(p\,|||\,q,\sigma)\xrightarrow{a}(p,\sigma)\,|||\,(q',\sigma')}(a\in\Sigma^{\Lambda\mathcal{B}\Omega\tau})$$

Figure 6.3: CSP's operational semantics

Figure 6.5 presents the rules of CSPs operational semantics updated with stores. For the details of the these rules refer to Chapter 5 page 119.

Figure 6.6 shows the updated and extended semantics of the parallel composition. The rules (par1,2,3) implement the normal behaviour of CSP parallel composition, but we update the rules so they will work with all soaCSP event types. The rules (par1,2) show that processes will independently evaluate any type of events outside the interface set. In (par3) the rule shows the behaviour of the parallel composition in case an event inside the interface set ($A\in\Sigma^{\mathcal{B}}$) is evaluated and this event is not communicating mobile channels, i.e. $a\notin\{c.(m)^{\varphi}\,|\,m\in \mathit{channels}(\Sigma)\}$. If the communicating data is a mobile channel then the rules (m-par1*) to (m-par6*) from Chapter 4 page 68 will be evaluated. Note that, $A$ (i.e. the parallel composition interface set) cannot contain service names or terminal events. Therefore, rules (SE), (parST1), (parST2), (parTses1), and (parTses2) define the behaviour of the parallel composition if these types of events need to synchronise for creating or terminating sessions. Whereas rules (parATses) and (parATses1) define the behaviour of the parallel composition if terminal events need to be independently evaluated.

In Figure 6.6 the rules (parST1) and (parST2) are an extended versions of (parST) from Chapter 3 page 49, which implements the *synchronised termination* of soaCSP,

(prefix) $\frac{}{(a \to p,\sigma) \xrightarrow{a} (p,\sigma)}\ (a \in \Sigma \wedge a \notin \{b.x, d.(c)^{\varphi} | x \in Z,\ c \in channels(\Sigma)\})$

(out1,2,3) $\frac{(e,\sigma) \xrightarrow{\tau} (e',\sigma)}{(a!e \to p,\sigma) \xrightarrow{\tau} (a!e' \to p,\sigma)}\ \frac{}{(a!\ell \to p,\sigma) \xrightarrow{\tau} (a!\sigma(\ell) \to p,\sigma)}\ \frac{}{(a!n \to p,\sigma) \xrightarrow{a.n} (p,\sigma)}\ n \in Z$

(ud.out1,2,3) $\frac{(e,\sigma) \xrightarrow{\tau} (e',\sigma)}{(a.e \to p,\sigma) \xrightarrow{\tau} (a.e' \to p,\sigma)}\ \frac{}{(a.\ell \to p,\sigma) \xrightarrow{\tau} (a.\sigma(\ell) \to p,\sigma)}\ \ell \in \sigma\ \frac{}{(a.n \to p,\sigma) \xrightarrow{a.n} (p,\sigma)}\ n \in Z$

(in) $\frac{}{(a?\ell \to p,\sigma) \xrightarrow{a.n} (p,\sigma[\ell \mapsto n])}\ n \in Z$ (ud.in) $\frac{}{(a.\ell \to p,\sigma) \xrightarrow{a.n} (p,\sigma[\ell \mapsto n])}\ \ell \notin \sigma \wedge n \in Z$

where in rules (out1,2,3), (ud.out1,2,3), (in), and (ud-in):
$(a \in \Sigma \wedge a \notin \{b, d.(c)^{\varphi} | b, d, c \in channels(\Sigma)\})$

(ch-out1) $\frac{}{(a!(m)^{\varphi} \to p,\sigma) \xrightarrow{\tau} (a!(\sigma(m))^{\varphi} \to p,\sigma)}$ (ch-out2*) $\frac{}{(a!(c)^{\varphi} \to p,\sigma) \xrightarrow{a.(c)^{\varphi}} (p,\sigma)}\ c \in channels(\Sigma)$

(ch-ud.out1) $\frac{}{(a.(m)^{\varphi} \to p,\sigma) \xrightarrow{\tau} (a.(\sigma(m))^{\varphi} \to p,\sigma)}\ m \in \sigma$

(ch-ud.out2*) $\frac{}{(a.(c)^{\varphi} \to p,\sigma) \xrightarrow{a.(c)^{\varphi}} (p,\sigma)}\ c \in channels(\Sigma)$

where in rules (ch-out1,2), (ch-ud.out1,2): $(a \in \Sigma \wedge a \notin \{b.x | x \in Z\})$, and $()^{\varphi}$ can be $()$, $()^{-}$, $()^{+}$, or $()^{-*}$; if $()^{\varphi}$ is $()^{-*}$, then $\alpha p := \alpha p - \{c\}$.

(ch-in*) $\frac{}{(a?(m)^{\varphi} \to p,\sigma) \xrightarrow{a.(c)^{\varphi}} (p,\sigma[m \mapsto c])}\ c \in channels(\Sigma)$

(ch-ud.in*) $\frac{}{(a.(m)^{\varphi} \to p,\sigma) \xrightarrow{a.(c)^{\varphi}} (p,\sigma[m \mapsto c])}\ m \notin \sigma \wedge c \in channels(\Sigma)$

where in rules (ch-in), (ch-ud.in):
$(a \in \Sigma \wedge a \notin \{b.x | x \in Z\})$, and $()^{\varphi}$ can be $()$, $()^{+}$, or $()^{-}$.

(var-ch1,2) $\frac{}{(m?INPC \to p,\sigma) \xrightarrow{\tau} (\sigma(m)?INPC \to p,\sigma)}\ \frac{}{(m!OUTC \to p,\sigma) \xrightarrow{\tau} (\sigma(m)!OUTC \to p,\sigma)}$

(var-ch3) $\frac{}{(m.OUTC \to p,\sigma) \xrightarrow{\tau} (\sigma(m).OUTC \to p,\sigma)}$

where in (var-ch1,2,3) $m$ is a mobile channel variable.

(asy-in1) $\frac{}{(a?{>}\ell \to p,\sigma) \xrightarrow{a \leftarrow .n} (p,\sigma[\ell \mapsto n])}$ (buffer-out1) $\frac{}{(a!{>}\ell \to p,\sigma) \xrightarrow{a \leftarrow .n} (p,\sigma)}$

(asy-out1) $\frac{}{(a!{<}\ell \to p,\sigma) \xrightarrow{a \rightarrow .n} (p,\sigma)}$ (buffer-in1) $\frac{}{(a?{<}\ell \to p,\sigma) \xrightarrow{a \rightarrow .n} (p,\sigma[\ell \mapsto n])}$

where in rules (asy-in1), (asy-out1), (buffer-out1), and (buffer-in1):
$(a \in \mathcal{B} \wedge a \notin \{b\_ < (c)^{\varphi}, d\_ > (n)^{\varphi} | c, n \in channels(\Sigma)\})$, and _ can be ? or !.

(asy-out2) $\frac{}{(a!{<}(m)^{\varphi} \to p,\sigma) \xrightarrow{a \rightarrow .m} (p,\sigma)}$ (buffer-in2) $\frac{}{(a?{<}(m)^{\varphi} \to p,\sigma) \xrightarrow{a \rightarrow .c} (p,\sigma[m \mapsto c])}$

(asy-in2) $\frac{}{(a?{>}(m)^{\varphi} \to p,\sigma) \xrightarrow{a \leftarrow .c} (p,\sigma[m \mapsto c])}$ (buffer-out2) $\frac{}{(a!{>}(m)^{\varphi} \to p,\sigma) \xrightarrow{a \leftarrow .m} (p,\sigma)}$

Where in rules (asy-in2), (asy-out2), (buffer-out2), and (buffer-in2):
$(a \in \mathcal{B} \wedge a \notin \{b\_ < x, d\_ > y | x, y \in Z\})$, and _ can be ? or !.

Figure 6.4: The operational semantics of the *prefix operator* of soaCSP

$$(\text{labelT})\frac{(p,\sigma)\xrightarrow{\omega}(p',\sigma)}{(L{:}p,\sigma)\xrightarrow{L.\omega}(f_p(L){:}p,\sigma)}\ (\omega\in\Omega)\quad(\text{labelS})\frac{(p,\sigma)\xrightarrow{\lambda}(p',\sigma)}{(L{:}p,\sigma)\xrightarrow{L.\lambda}(L{:}p',\sigma)}\ (\lambda\in\Lambda)$$

$$(\text{labeli})\frac{(p,\sigma)\xrightarrow{\tau}(p',\sigma)}{(L{:}p,\sigma)\xrightarrow{\tau}(L{:}p',\sigma)}$$

$$(\text{def})\frac{}{(N,\sigma)\xrightarrow{\tau}(p,\sigma)}\ (N=p\in\mathcal{D})\quad(\text{def2})\frac{}{(N,\sigma)\xrightarrow{\tau}(N\Rightarrow p,\sigma)}\ (N\Rightarrow p\in\mathcal{D})$$

$$(\text{ps-def})\frac{}{(*N,\sigma)\xrightarrow{\tau}(*(N\Rightarrow p),\sigma)}\ (N\Rightarrow p\in\mathcal{D})\ (\text{srv-def1})\frac{}{(N\Rightarrow p,\sigma)\xrightarrow{N\top}((new)\,(N\Rightarrow p),\sigma)}$$

$$(\text{srv-def2})\frac{}{((new)\,(N\Rightarrow p),\sigma)\xrightarrow{N\top l}(\underline{l}{:}p,\sigma)}\ (l\notin\mathbb{L},\ \mathbb{L}:=\mathbb{L}\cup\{l\};\ \Sigma^{\Lambda\mathcal{B}\Omega}:=\Sigma^{\Lambda\mathcal{B}\Omega}\cup\mathbb{L}:\Sigma^{\Lambda\mathcal{B}\Omega})$$

$$(\text{ps-wind})\frac{}{(*(N\Rightarrow p),\sigma)\xrightarrow{\tau}((N\Rightarrow p),\sigma)\parallel(*(N\Rightarrow p),\sigma)}$$

$$(\text{srv-inv})\frac{}{(N\Leftarrow\{q\},\sigma)\xrightarrow{N\perp}(\underline{l}{:}q,\sigma)}\ l\in\mathbb{L}\quad(\text{joinS})\frac{}{(N\Leftarrow^{+}\{q\},\sigma)\xrightarrow{N+}(\underline{l}{:}q,\sigma)}\ (l\in\mathbb{L})$$

$$(\text{listen})\frac{}{(LISTEN,\sigma)\xrightarrow{\omega}(STOP,\sigma)}\ (\omega\in\{\surd,\dagger\})\ (\text{T-hdr1})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\blacktriangleright q,\sigma)\xrightarrow{a}(p',\sigma')\blacktriangleright(q,\sigma)}\ (a\in\Sigma^{\Lambda\mathcal{B}\tau})$$

$$(\text{T-hdr2})\frac{(p,\sigma)\xrightarrow{\dagger}(p',\sigma)}{(p\blacktriangleright q,\sigma)\xrightarrow{\tau}(q,\sigma)}\quad(\text{T-hdr3})\frac{(p,\sigma)\xrightarrow{\surd}(p',\sigma)}{(p\blacktriangleright q,\sigma)\xrightarrow{\surd}(p',\sigma)}$$

Figure 6.5: The CSPs part of the soaCSP operational semantics

in the same way as we explained it for CSPa, see Chapter 3 page 49. The *synchronised termination* replaces the *distributed termination* of the standard CSP. The distributed termination of the standard CSP is implemented by (ParT1,2,3) in Figure 2.2 page 42.

Finally, we remark that channels in soaCSP, as in standard CSP, can carry simultaneously multi-components data in both directions. For instance, if $a$ is a channel name then the event $(a?x!y!z)$ is a valid event which accepts an integer and sends two integers. In the operational semantics, we give rules for channels that carry only one data component; this can be extended to allow the generalised form of events, i.e., $a?INPC$, $a!OUTC$, or $a.OUTC$.

## 6.3 The relationship between BPEL and soaCSP

As we stated in the Introduction chapter, although process calculi are equipped with formal semantics, which enables formal verification of system properties, a feature not available in the semi-formal modelling languages of the internet standards set (WS-*), the latter languages are still the dominant modelling languages. Therefore, in this section, we discuss the relationship between our calculus and these standards. In particular, we discuss the relationship between our calculus and the BPEL orchestration language from the (WS-*) set; for the description of the BPEL language

$$(\text{par3})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma') \quad (q,\sigma)\xrightarrow{a}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a}(p',\sigma')\parallel_A(q',\sigma'')}\ a\in A\,,\ a\in\Sigma^{\mathcal{B}}\,,\ a\notin\{d.(c)^{\varphi}|c\in channels(\Sigma)\}$$

$$(\text{par1})\frac{(p,\sigma)\xrightarrow{a}(p',\sigma')}{(p\parallel_A q,\sigma)\xrightarrow{a}(p',\sigma')\parallel_A(q,\sigma)}\ a\notin A, a\in\Sigma^{\mathcal{B}\tau}\quad (\text{par2})\frac{(q,\sigma)\xrightarrow{a}(q',\sigma')}{(p\parallel_A q,\sigma)\xrightarrow{a}(p,\sigma)\parallel_A(q',\sigma')}\ a\notin A, a\in\Sigma^{\mathcal{B}\tau}$$

$$(\text{m-par1*})\frac{(p,\sigma)\xrightarrow{a.(c)}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)}(p',\sigma')\parallel_A(q',\sigma'')}\quad (\text{m-par3*})\frac{(p,\sigma)\xrightarrow{a.(c)}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)^+}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)^+}(p',\sigma')\parallel_{A'}(q',\sigma'')}\ A'=A\cup\{c\}$$

$$(\text{m-par2*})\frac{(p,\sigma)\xrightarrow{a.(c)^+}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)^{\varphi}}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)^+}(p',\sigma')\parallel_{A'}(q',\sigma'')}\ A'=A\cup\{c\}, \text{ and } ()^{\varphi} \text{ can be } () \text{ or } ()^+$$

$$(\text{m-par4*})\frac{(p,\sigma)\xrightarrow{a.(c)}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)^{\varphi}}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)^-}(p',\sigma')\parallel_{A'}(q',\sigma'')}\ A'=A-\{c\}, \text{ and } ()^{\varphi} \text{ can be } ()^-, \text{ or } ()^{-*}$$

$$(\text{m-par5*})\frac{(p,\sigma)\xrightarrow{a.(c)^-}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)^{\varphi}}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)^-}(p',\sigma')\parallel_{A'}(q',\sigma'')}\ A'=A-\{c\}, \text{ and } ()^{\varphi} \text{ can be } (), ()^-, \text{ or } ()^{-*}$$

$$(\text{m-par6*})\frac{(p,\sigma)\xrightarrow{a.(c)^{-*}}(p',\sigma') \quad (q,\sigma)\xrightarrow{a.(c)^{\varphi}}(q',\sigma'')}{(p\parallel_A q,\sigma)\xrightarrow{a.(c)^-}(p',\sigma')\parallel_{A'}(q',\sigma'')}\ A'=A-\{c\}, \text{ and } ()^{\varphi} \text{ can be } (), ()^-, \text{ or } ()^{-*}$$

where in rules (m-par1*) to (m-par6*):
$a\in A\,,\ a\in\Sigma^{\mathcal{B}}\,,$ and $a\in\{d.(c)^{\varphi}|c\in channels(\Sigma)\}$

If $(\lambda,\lambda'\in\Lambda)$, $(l,L,L'\in\mathbb{L})$, and the binary operation $([L.]\lambda,[L'.]\lambda')\to SL$ is defined by the following table:

(SL-table)

| $[L.]\lambda$ / $[L'.]\lambda'$ | $[L.]N\top l$ / $N\bot$ | $[L.]N\top l$ / $L'.N\bot$ | $[L.]N\top l$ / $N+$ | $[L.]N\top l$ / $L'.N+$ |
|---|---|---|---|---|
| $SL$ | $l$ | $L':l$ | $l$ | $L'$ |

then:

$$(\text{SE})\frac{([L:]p,\sigma)\xrightarrow{[L.]\lambda}([L:]\underline{l}:p',\sigma) \quad ([L':]q,\sigma)\xrightarrow{[L'.]\lambda'}([L':]\underline{l}:q',\sigma)}{([L:]p\parallel_A[L':]q,\sigma)\xrightarrow{\tau}(([L:]\underline{l}:p')[\![SL/[L:]\underline{l}]\!]\parallel_{SL:A}([L':]\underline{l}:q')[\![SL/[L':]\underline{l}]\!],\sigma)}\ ([L.]\lambda,[L'.]\lambda')\in \text{(SL-table)}$$

$$(\text{parST1})\ \frac{(p,\sigma)\xrightarrow{\surd}(STOP,\sigma) \quad (q,\sigma)\xrightarrow{\omega}(STOP,\sigma)}{(p\parallel_A q,\sigma)\xrightarrow{\surd}(STOP,\sigma)}\ \omega\in\{\surd,\dagger\}$$

$$(\text{parST2})\ \frac{(p,\sigma)\xrightarrow{\dagger}(STOP,\sigma) \quad (q,\sigma)\xrightarrow{\dagger}(STOP,\sigma)}{(p\parallel_A q,\sigma)\xrightarrow{\dagger}(STOP,\sigma)}$$

$$(\text{ParTses1})\frac{(L{:}p,\sigma)\xrightarrow{L.\surd}(f_p(L){:}p,\sigma) \quad (L{:}q,\sigma)\xrightarrow{L.\omega}(f_p(L){:}q,\sigma)}{(L{:}p\parallel_A L{:}q,\sigma)\xrightarrow{L.\surd}(f_p(L){:}SKIP,\sigma)}\ \omega\in\{\surd,\dagger\}$$

$$(\text{ParTses2})\frac{(L{:}p,\sigma)\xrightarrow{L.\dagger}(f_p(L){:}p,\sigma) \quad (L{:}q,\sigma)\xrightarrow{L.\dagger}(f_p(L){:}q,\sigma)}{(L{:}p\parallel_A L{:}q,\sigma)\xrightarrow{L.\dagger}(f_p(L){:}LISTEN,\sigma)}$$

$$(\text{ParATses1})\frac{(L{:}p,\sigma)\xrightarrow{L.\omega}(f_p(L){:}p,\sigma) \quad ([L':]q,\sigma)\xrightarrow{[L'.]\omega'}(q',\sigma)}{(L{:}p\parallel_A[L':]q,\sigma)\xrightarrow{L.\omega}(f_p(L){:}p\parallel_A[L':]q,\sigma)}\ \omega,\omega'\in\{\surd,\dagger\}$$

$$(\text{ParATses})\frac{([L:]p,\sigma)\xrightarrow{[L.]\omega}(p',\sigma) \quad (L':q,\sigma)\xrightarrow{L'.\omega'}(f_p(L'):q,\sigma)}{([L:]p\parallel_A L':q,\sigma)\xrightarrow{L'.\omega'}([L:]p\parallel_A f_p(L'):q,\sigma)}\ \omega,\omega'\in\{\surd,\dagger\}$$

Figure 6.6: The operational Semantics of soaCSP's parallel composition

refer to Chapter 2 page 13. This is because, from the design perspective, soaCSP is considered to be an orchestration calculus, because services gain control over other services by triggering their execution [14].

We study this relationship by discussing the possibility of encoding BPEL into soaCSP; a detailed encoding to facilitate a model transformation from BPEL specifications to soaCSP models are proposed as future work. As BPEL lacks formal semantics, the encoding is performed by informally showing that the syntax primitives of BPEL are encodable in soaCSP.

As mentioned in Section 2.4.2, BPEL specifications generally have three main parts: *partners*, *variables*, and *control flow* activities and *exception handlers*. Partners and variables are necessary in the BPEL to define the dependencies between services in a service composition. However, in soaCSP these dependencies are defined in the semantics. Therefore, we compare the two languages only with respect to their control flow constructs.

We consider the BPEL control flow constructs listed in Section 2.4.2 page 34, and we abstract from these constructs the attributes which require implementation details like operations, name spaces and similar attributes. Additionally, recall that in this thesis we focus on the so-called control part of BPEL, therefore, constructs dealing with XML data and XML schema are not considered.

The encoding is as follows:

- < `process` > construct:

  In BPEL

  ```
  <process name= "service1"
  ......................
  /process>
  ```

  This defines a BPEL specification for a service. In soaCSP, services can be defined as: *service1* $\Rightarrow$ .....

- < `receive` > construct:

  In BPEL

```
<receive name= "receiveInput"
 partnerLink = "client"
 portType = "chName"
 Variable="clientMSG"
.....................
createInstance= "yes/no" /receive>
```

In soaCSP, this message is considered as input message: *chName?clientMSG*. The activity name and the partner name are not required in soaCSP as they are determined by the calculus semantics. In soaCSP, the `createInstance` attribute is encoded by attaching $*$ to the service name and not upon receiving messages, i.e., **service1* $\Rightarrow$ .....

Information about service instances, as specified in Section 2.4.2, is maintained through $<$ `correlationSets` $>$. In soaCSP, correlation sets are not required as state information is maintained by the calculus semantics.

In BPEL, *chName* is the name of the port in the WSDL interface of the service. As mentioned previously, the BPEL specification uses the WSDL interface to import and export information. Therefore, ports here serve as communication channels. As a result, in this encoding, *portType* is always encoded into a channel name.

- $<$ `reply` $>$ construct:

  In BPEL

```
<reply partnerLink="client"
 portType="chName"
 variable="response"
 ......................
 messageExchange="clientMSG"  /reply>
```

If the message exchange pattern is **one-way** only, that is, the message is sent in one direction, and no response is required [67], then the `receive` activity does not need any other activity. However, if the message received by the

< `receive` > activity needs a response, then the message exchange pattern is **request-response**, that is, the input message needs an immediate response.

The < `reply` > activity is used if the received message in the < `receive` > activity needs a response. In BPEL, if the input and output messages use the same port name (defined in WSDL), then this communication is synchronous, and the activity will be suspended until the response is received. In soaCSP, this message exchange can be encoded as *chName?clientMSG!response* or *chName?clientMSG* $\rightarrow$ *chName!response* as long as the *chName* is in the interface set of the parallel composition between these two services. The difference between the two encodings is that the former requires the response message to be sent at the same time as the input message is received, whereas in the latter case response is not required, but the service will be block until a response is received.

In BPEL, if the communication should be asynchronous, that is, the service should be able to continue its execution until a response is received, then the input and the output messages should use two different `portType`s. Achieving asynchronicity by using two different channel names is not necessary in soaCSP. soaCSP supports buffered asynchronous communications by using ?> instead of ? and !< instead of !, so changing the channel names in asynchronous communications is not required. However, asynchronous communications which are implemented as two synchronous communications via two different channels, as in the BPEL communication, is also permitted by the calculus.

The < `messageExchange` > attribute explicitly defines the input message that requires this response. In soaCSP, this should be clear from the semantics, but if needed then the communication *chName?clientMSG!response* binds the input and the output.

- < `invoke` > construct:

  In BPEL

```
<invoke  name="invokeSer"
partnerLink="service2"
 portType = "chName"
 inputVariable="VariableName1"
```

```
 outputVariable="VariableName2"
 ..............
</invoke>
```

This defines an invocation to another service. In soaCSP, this can be encoded as: *service2* $\Leftarrow \{chName?VariableName1!VariableName2 \rightarrow \ldots\}$, or *service2* $\Leftarrow \{\ chName?VariableName1 \rightarrow chName!VariableName2\ \}$, because the communication here is synchronous as the input/output messages use the same port.

If asynchronous communication is required then the < `invoke` > activity only sends the request and the response message should be received in a following *receive* activity.

We use the partner name here not to determine the dependencies as this is achieved by the semantics, but it is used to trigger the labelling algorithm in the semantics which replaces the correlation sets in BPEL.

- < `assign` >, < `copy` >, < `from` >, and < `to` > constructs:

  Generally speaking, soaCSP, like CSP, does not have assignment primitives. Variables' values can be changed upon receiving new data through an inputting channel.

- < `validate` > construct:

  The validation of messages can be carried out as a part of the service logic.

- < `wait` > Construct:

  soaCSP does not support time. Therefore, activities related to time are not supported.

- < `extensionActivity` > construct:

  < `extensionActivity` > construct can be directly encoded in soaCSP as process callings inside the body of the current service.

- < `sequence` > construct:

< `sequence` > construct can be directly encoded in soaCSP as a sequential composition (;).

- < `flow` > construct:

  < `flow` > construct can be directly encoded in soaCSP as a parallel composition $\|_A$. If < `link` > elements are used within the `flow` activity, then these links should be included in the interface set (i.e., $A$) of the parallel composition.

- < `if` >, < `else` >, < `elseif` >, and < `switch` > constructs:

  Each of these constructs can be encoded into an if-then-else construct. As stated in the Introduction chapter soaCSP is founded on CSP, and as stated in Chapter 2, in CSP we assume that functions defining set, conditional blocks, and loops are defined.

- < `forEach` >, < `while` > and < `repeatUntil` > constructs:

  < `while` > and < `repeatUntil` > constructs can be encoded into a while construct. As stated in the Introduction chapter soaCSP is founded on CSP, and as stated in Chapter 2, in CSP we assume that functions defining set, conditional blocks, and loops are defined. However, we formally define the latter two constructs in Chapter 8.

  For encoding the `forEach`, we use if-then-else, recursion and counter process as follows:

```
Counter(i)= dec!(i-1) → Counter(i-1)
forEachBody(p)=  dec?i → ( if (i  != 0)
                                then (p; forEachBody(p)) else SKIP )
forEach(n,p) = forEachBody(p) ||{| dec |}  Counter(n+1)
```

- < `pick` > construct:

  < `pick` > construct can be encoded into soaCSP using the external choice $\Box$ as follows:

  $(event1 \rightarrow p)\Box(event2 \rightarrow p)\Box \ldots$, where $p$ represents the activity associated with the < `pick` > element.

In BPEL, the specification of the < `eventHandler` > element is similar to < `pick` > except that events in < `eventHandler` > could be alarms. Alarms are a representation of time and soaCSP does not support time. Therefore, in soaCSP, the < `eventHandler` > element is encoded in the same way as the < `pick` > element.

- < `scope` > construct:

  Sub-dividing the service scope into different scopes can be achieved by dividing the service code into different sub-processes. As in soaCSP, each process defines a new scope (i.e. in the semantics each process has a different store).

- < `throw` >, < `rethrow` >, and < `faultHandler` > constructs:

  soaCSP is not equipped with an exception system (an exception system is proposed in Chapter 8). However, a throw can be defined in soaCSP as an explicit process and be managed by the designer as follows:

  $\texttt{throw= error} \rightarrow \texttt{STOP}$
  $\texttt{SCOPE = MainActivity} \parallel_{\{\mid error \mid\}} \texttt{(error} \rightarrow \texttt{faultHandler)}$

- < `compensate` >, < `compensateScope` >, and < `compensationHandler` > constructs:

  As mentioned above, soaCSP is not equipped with an exception system. However, in BPEL, the compensations handler is a predefined fixed activity. Therefore, compensations can be initiated in soaCSP by an explicit event and managed by the designer as follows:

  $\texttt{compensate= startC} \rightarrow \texttt{STOP}$
  $\texttt{SCOPE = MainActivity} \parallel_{\{\mid startC \mid\}} \texttt{(startC} \rightarrow \texttt{compensateHandler)}$

- < `exit` > and < `terminationHandler` > constructs:

  In soaCSP, < `exit` > can be encoded into one of the following three processes:

  If < `exit` > is a signal indicating that the process has finished then this is equivalent to *SKIP*. Alternatively, if < `exit` > is initiated because of another

event, then in soaCSP, this can be encoded as we did with $<$ `throw` $>$, where we can encode a termination handler in the same way.

In soaCSP, a process can choose to terminate because another service has finished in the composition by using *LISTEN* process. In this case a termination handler can started by using ▶ operator.

- $<$ `empty` $>$ construct:

  $<$ `empty` $>$ construct can be encoded as *STOP* in soaCSP.

## 6.4 Conclusions and Related Work

This chapter presents soaCSP as the final theoretical model in this thesis. soaCSP is presented as the result of combining the previous three calculi: CSPa from Chapter 3, $\mathcal{M}$CSP from Chapter 4, and CSPs from Chapter 5.

We develop a model in which these three calculi can work. The compound calculus draws interesting results on orchestrating multiparty sessions. In soaCSP, a notion of temporarily merging sessions follows from labelling mobile channels. Hence, mobile channels carry the state of a session with them, which facilitates communicating with this session without being part of the service composition. This gives an easy solution for services in compositions to directly connect to legacy systems.

This is different from the permanent session merging in $\mu se$ calculus [45], where sessions can merge dynamically. This can be achieved in soaCSP if the communicated name is used in all communications in the receiver service.

Session merge as presented in this chapter is similar to the work in [28], as both calculi delegate services by delegating channel names. However, the way that soaCSP deals with session is different from [28]. In soaCSP, channels are the communication primitives in the calculus and session keys are dynamically tagged to channels during the process of sessions creation. Therefore, the designer, in soaCSP, is aware of the merge but not managing it. Whereas in [28] channels represent the sessions keys. Therefore, in [28], the designer is the one who initiates the merge and manages the session names.

The work in [110] is similar to the work in [28]; however, the merge is achieved at the choreography level by delegating session names to achieve choreography com-

positions rather than delegating services. In [110], the designer is also the one who manages the session names.

In soaCSP, processes can be defined in addition to services, therefore the communication between ordinary processes and sessions can be modelled in soaCSP (see Chapter 7 for an example).

To the best of our knowledge, soaCSP is the first calculus to admit synchronous, asynchronous, interleaving and mobile communications in multiparty sessions.

The chapter finishes by encoding the control part of the BPEL into soaCSP. As been shown in the encoding, soaCSP is expressive enough to encode the control part of BPEL. Therefore, soaCSP could be used to verify this part of BPEL language. A transformation from BPEL models to soaCSP models could be useful to facilitate this verification; we propose it as future work.

In addition to the control part in BPEL, the language has a number of operations dealing with XML data and XML Schema and their parsing. For instance, the element ($<$ `import` $>$) which is used to declare a dependency on external XML Schema or WSDL definitions is not considered in soaCSP.

Moreover, timing operators in BPEL like alarms and timeout events are not encodable in soaCSP because our calculus does not support time.

# 7

# Model Checker for soaCSP

## 7.1 Introduction

In Section 2.1 we carried out a survey of the state-of-the-art of academic research in the field of service-oriented computing. A considerable amount of this research was conducted to support SOC modelling languages with formal semantics, and so to be able to reason on the correctness of these models. As already mentioned, we focus in this thesis on process calculi, and we observe that although previous SOC calculi are equipped with formal semantics, a feature not available in the WS standards and which enables formal verification of properties, internet standards are still the dominant modelling languages.

In our view, this is mostly because formal methods are often perceived as hard to apply. Therefore, to support the practical use of our calculus we have encoded soaCSP in the model checker of CSP (FDR [12]).

FDR implements the mathematical machinery and the theory of refinement that Hoare built for reasoning on the external behaviour of systems; see Chapter 2. It provides simple proof techniques to assert the conformance between specification and implementation, deadlock-freedom, and divergence-freedom, in addition to determinism and bisimulations (we illustrate some of these features in the case study in Section 7.3).

Other model checkers, such as SPIN [85], which accepts models in Promela, or Maude [22], could be used if these checks are implemented.

The model checker PAT [131] supports CSP models and implements CSP refine-

ment checks. However, PAT does not support the generalised version of CSP parallel composition. Therefore, we prefer FDR where this version of the parallel composition is implemented.

**Contributions in this chapter**

1. The implementation of soaCSP in $\text{CSP}_M$ [128] (the machine readable version of CSP). As a result, all CSP tools which use $\text{CSP}_M$ as their input language, could be used to reason on the correctness of soaCSP models; details follow in the chapter.

2. Encoding a finance case study in soaCSP, and illustrating the usability of soaCSP.

**Structure of this chapter** Section 7.2 presents the implementation of soaCSP in $\text{CSP}_M$. Section 7.3 illustrates the usability of our calculus by encoding the finance case study in soaCSP. Finally, Section 7.4 concludes the chapter and discusses the related works.

## 7.2 Implementing soaCSP in FDR

In this section, we provide an implementation of soaCSP in FDR [12] which is the model checker of CSP. FDR uses $\text{CSP}_M$ [128] as the input language. $\text{CSP}_M$ is the machine readable version of CSP. Encoding soaCSP into $\text{CSP}_M$ permits soaCSP users to use other CSP tools, like the trace animator Probe [7].

As previously mentioned, soaCSP consists of three parts: CSPa, $\mathcal{M}$CSP, and CSPs. In the following sections we discuss the implementation of each part in $\text{CSP}_M$.

### 7.2.1 Implementing CSPa in FDR

To implement CSPa in FDR we use the encoding statements (listed in Definition 3.5). We also encode $\text{B}_\Sigma$ to be the CSP process defined in Section 3.2. We use *in* and *out* keywords in front of channel names to encode $a \leftarrow$ and $a \rightarrow$ events respectively. Thus, $in.a?x$ and $out.a!x$ represent $a?{>}x$ and $a!{<}x$ events respectively in the theoretical model.

```
bufferedEvents =  let
       bf(a,s) = if null(s) then ( (out.a?x -> bf(a,<x>)) [] SKIPP )
                      else ((in.a!head(s) -> bf(a,tail(s))[]
                           #s<N & (out.a?x -> bf(a,s^<x>))) [] SKIPP)
                  within (||| x:ev @ bf(x,<>))
```

According to our model this buffer should be created for all events in $\Sigma$. However, to reduce the memory consumption we create this buffer for events in *ev* only, where *ev* is the set of asynchronous events provided by the user. If the user is happy to include all events then set *Events* can be used; *Events* in $\text{CSP}_M$ corresponds to $\Sigma$ in CSP. All the buffers start empty.

In our implementation, we prefer to design the buffered-$\Sigma$ to be bounded and for a set of events instead of the whole $\Sigma$. This is to reduce the consumption of memory and accelerate the assertion time in FDR. Although these limits are not requirements of FDR we prefer to give the user the ability to accelerate the assertion time and reduce memory consumption.

The preprocessing step in our theoretical model, which installs the buffered-$\Sigma$, is encoded in our implementation by the function *asy*. This function puts the system process in parallel with the buffered-$\Sigma$.

```
asy(p)= p  [| {|in.a, out.a, term | a<-ev |} |]   bufferedEvents
```

To deal with termination, users should use *SKIPP* instead of *SKIP* to evaluate the new terminal signal *term* first, and use *par* function instead of $\|_A$ to force synchronisation on successful terminations of processes. Synchronisation is achieved through *term*.

```
SKIPP = term -> SKIP
par(p,A,q)= p  [| union({|term|},A) |]  q
```

If a system terminates the buffers will be forced to terminate due to the external choice with *SKIPP*. This choice is resolved by the environment. Thus, the buffered-$\Sigma$ will terminate as soon as the system terminates. This will avoid the problem of dangling buffers when the system terminates.

### 7.2.2 Implementing $\mathcal{M}$CSP in FDR

To implement $\mathcal{M}$CSP in FDR we use some of the encoding ideas listed in Section 4.4, in particular the regulator process. $\mathcal{M}$CSP relies on the idea of updating the parallel composition's interface set; however, FDR does not support updating the interface set. In FDR, the interface set is computed at the beginning then processes are evaluated accordingly. To overcome this restriction we change the parallel composition to a parallel composition with regulator. In other words, if $p, q$ are $\mathcal{M}$CSP processes then $p \parallel_A q$ will be written as follows:

$(reg(A)|||p) \parallel_\Sigma (reg(A)|||p)$, where $reg(A) = \Box_{x:\Sigma-A}\ x \rightarrow reg(A)$.

As a result, participants in the parallel composition will synchronise on all events; if the event is in $A$ then it should be evaluated by the processes, otherwise events will be passed by the regulator. The clue here is that the set $A$ in the regulator can be updated.

To update the set $A$ in the regulator we define a special channel *chUpdate* which carries the mobile channel and a mark. If the mark is *plus* then add the mobile channel to the set $A$, so the regulator will no longer pass it out from the parallel composition and the process should evaluate it. If the mark is *minus* then omit the mobile channel from the set $A$ , so the regulator will be able to pass it out from the parallel composition.

*chUpdate* is defined in FDR as follows:

```
datatype Marks = plus | minus
channel chUpdate: Marks.Mobch
```

where *Mobch* is the set of all mobile channels in the system (provided by the user). In our implementation, mobile channels refer to the name part of channels without their type definitions. However, in FDR the channel's name is attached to its type so the name cannot be used without its data part. Therefore, we ask the user to define aliases for mobile names. The aliases will be related to the actual channel name by using the *ch* function. *Mobch* will include channel aliases rather than actual channel names. The following example illustrates how these definitions can be used if we want to send channel *product* as a mobile channel.

```
channel mproduct

Mobch={|mproduct|} -- aliases for mobile channels
```

```
ch(mproduct)= product
```

Sending a mobile channel is done in two steps: first we send the name through the carrying channel, then we evaluate the *chUpdate* which will update the regulator according to the marks used. If the channel has no marks then the second step should be skipped. For instance, if we have two processes defined in $\mathcal{M}$CSP as $p = COM!(product)^+ \rightarrow product?z \rightarrow p$ and $q = COM?(x)^+ \rightarrow x?y \rightarrow q$, then in our model these processes should be written as:

```
p= COM!mproduct -> chUpdate!plus!mproduct -> product?z -> p
q= COM?x -> chUpdate!plus!x -> ch(x)?y -> q
```

If we want the process to work in a mobile environment (i.e. sending/receiving channels), then we should prepare the process for that by using the *Mobile* function. This function will attach the process to a regulator then let them synchronise on *chUpdate*, so the process can trigger changes in its regulator through channel *chUpdate*. Finally, we hide the communications from the environment through *chUpdate* to avoid mixing updates of different processes working in parallel.

```
Mobile(p,A)= (reg(A) [| {|chUpdate|} |] p) \{|chUpdate|}
```

We define the regulator in FDR as follows:

```
reg(A) = ([] x:{| y | y<-diff(diff(Events,A),{|chUpdate|})|} @ x -> reg(A))
          [] (chUpdate?dir?x -> if (dir==plus) then reg(union(A,{|ch(x)|}))
                                else reg(diff(A,{|ch(x)|}))
```

Finally, we implement the parallel composition in FDR as follows:

```
parmob(p,q)= (p [| diff(Events,{|chUpdate|}) |] q)
```

The two processes synchronise in all events except *chUpdate*, which is a private communication between each process and its regulator.

### 7.2.3 Implementing CSPs in FDR

In this section, we show how to implement CSPs into FDR.

To facilitate the encoding we assume the following:

- the number of parallel instances in the definition of persistent services is bounded by an integer number $pn$ (set by the designers).
- the levels of nested sessions is bounded by an integer number $sd$ (set by the designers).

We first start by implementing a label generator. In the generator we fix the pattern of labels to be ($l.N$, where $N > 0$). A chain of labels will be written as ($l.0.l.1.\ldots.l.n$).

```
labelGen(i) = new!(i+1) ->  labelGen(i+1)
```

We implement the services as follows:

- Published service: $publish(p, N) = \ldots$, where $p$ is the process and $N$ is the name given for this published process. This is equivalent to $N \Rightarrow p$.
- Persistent service: $perServ(p, N) = |||x : \{0..pn\} @ publish(p, (N.x))$

  This is equivalent to $(*N)$ with bound $pn$. $|||$ is used because in CSP $||| = \|_{\{\}}$

  Now, the name of the published service will be the given name ($N$) with an index, e.g. $N.0$ will be the name of the first copy.
- Invocation service $inv(q, N, A) = \ldots$, where $q$ is representing the client protocol, $N$ is the name of the published service needed, and $A$ is the set of events which this service wants to synchronise on with other services. This is equivalent to $(N \Leftarrow \{q\}) \|_A$.

We define the published service as:

```
publish(p,N) = diamonds(new)?i ->
             if member(N,{z.z' | z<-pnames, z'<-V})
                then diamonds(slabel(session,N))!i -> SKIP
                else diamonds(slabel(session,x))!(0.i) -> SKIP


slabel(x,i) =  x.i    -- Session name
```

The published service will behave as follows:

1. It will issue a label from the label generator, where $diamonds(new)$ denotes unsessioned communication with the generator.
2. Then this label will be passed to the invocation service through the unsessioned channel $session.N$, where $N$ is the name of the published service. Here, we distinguished

between *session.N.index.i* of the persistent service and *session.N*.0.*i* of a regular service. This can be done by examining the name of the service (checking whether it has an index or not).

3. Finally, the process $p$ will terminate successfully.

Recall that an unsessioned channel has no label (communication outside the boundaries of a session).

We define the invocation service as:

```
inv(q,N,A) = diamonds(slabel(session,N))?((x.y)@@i) ->
                  (((label(l.y,q) [| {| cl |}|]  level)\{|cl|})
                     [| union({|close|},{|l.y.G | G<-A |}) |]
  ((label(l.y,process(N)) [| {| cl |}|]  level)\{|cl|})  )
```

The invocation service will behave as follows:

1. It accepts the label from the published service of name $N$ through the unsessioned channel *session.N*, where $N$ is the name of the published service. Here, we extract the label $y$ from $i$ leaving the index or 0.

2. Then, it evolves to a parallel composition containing the process $q$ and the published process with name $N$. The published process can be retrieved by using the function *process(N)*. The two processes will synchronise on the set of events $A$ which is provided by the user and on *close* which is the explicit close of sessions.

   *Note.* In our model, the relationship between a process and its published names should be explicitly defined via the function *process(servicename)=processname*. For instance, *process(Bank)=Bankpro* is defining the name *Bank* as the published name for the process *Bankpro*.

3. Finally, the parallel composition will be given the label $l.y$ using the function *label(L,pro)*.

The join invocation works in the same way except that the label, which will be given, contains the key *js*, and the function used for labelling is *labelj(L,pro)*.

The labelling operator is not implemented in FDR. For that we implement a function which does the labelling by renaming all the names inside a process to *label.channel*.

```
label(L,pro) = pro [[ (cl.ll)<-(cl.(<L>^ll)) , (js.lab.x)<-(L.x) ,
           x<-(L.x) , (diam.x)<-(diam.x), (fp.x)<-x
|  x<-diff(Events,{|cl,session|}), lab <-labonly ]]
```

where *pro* is the process and $L$ is the label. Here, *diam* and $fp$ are used as keys to control the labelling.

*diam* events always skip the label renaming whereas $fp$ events skip the labelling only once in the current renaming operation.

*diam* and $fp$ keys are installed using the functions:

```
diamonds(s) = diam.s
up(s)= fp.s
```

Which represent the events $a\diamond$ and $a\uparrow$ respectively.

The key *js* indicates that the process is joining the current session instead of its own session. Therefore, the previous label *js.lab* is replaced by the current label $L$.

The *cl* event is used by the termination algorithm. *cl* stores the current label $l$ to use this label in the hierarchy termination.

*labelj* is a variation from the *label* function, which keeps the extra session communications with no change and does not store the current label in the termination variable *cl*.

```
labelj(L,pro) = pro [[  (js.lab.x)<-(L.x) ,
 x<-(L.x) , (diam.x)<-(diam.x) ,  (fp.x)<-(fp.x)
|  x<-diff(Events,{|cl,session|}), lab <-labonly , ll<-eeh1]]
```

The label operator ends with an outer renaming *extracom(pro)* which removes the extra session keys *diam,fp*. In addition, if a join invocation reaches this upper level and has not been renamed before then this join invocation is not nested and should create a new session. Therefore, this renaming keeps the join invocation label and just removes the $js$ key.

```
extracom(pro) = pro [[ (js.leb)<-(leb)  , (diam.x)<-x , (fp.x)<-x
                     | x<-Events, leb <-EWL ]]
```

The termination algorithm is supported with these functions:

```
SSKIP= cl!<> -> SKIP
level= cl?s->levelup(cl.s)
levelup(cl.s)= if ((s)==(<>)) then SKIP else head(s).close ->
                                              levelup(cl.tail(s))
LISTEN= (close -> STOP) [] SKIP
term_Hnr(p,q)= (p [| {| close |} |] (close -> q))
```

As a result of not been able to change *SKIP* code, because it is part of the FDR code, we ask designers to use *SSKIP*, which will tag *SKIP* with labels.

Moreover, ordinary *SKIP* is represented in our implementation by the event *close*.

*SSKIP* will initiate the termination algorithm by sending the termination variable *cl* to the levelling up engine *level,levelup*.

*cl* is a variable which is used to store all the labels that this terminated process passed by. FDR does not support variables with storage, therefore, we attached to *cl* an empty list and accumulatively store the labels in this sequence during the renaming phase. Labels should be stored in order.

The levelling up engine then uses the stored labels in *cl* to start the termination algorithm. This is done by evaluating the event *close* with every label in *cl* list in backward order.

Finally, the parallel composition between a published service and an invocation service $N \Rightarrow p \parallel_A N \Leftarrow \{q\}$ , in our implementation, should be written as:

```
extracom(publish(p,N))  [| {|session,close|} |]  extracom(inv(q,N,A))
```

This parallel composition only synchronises a published service with an invocation. In case two published services or two invocations are put in parallel, they will not synchronise. This agrees with our operational semantics.

To allow the system to work, the parallel composition operator should be put in parallel with the label generator.

```
... [|{| new |}|]  labelGen(0)
```

## 7.3 Case study

In this section, we provide a detailed implementation for the Finance Case Study [65, 66, 132] which was created within the scope of the Sensoria EU project [8]. First we present the informal specification of the case study then we discuss the implementation and the evaluation of the case study using the soaCSP functions in CSP tools.

### 7.3.1 Finance Case Study

We consider a bank portal where the client requests a credit. The credit request scenario is described as follows:

The client first logs into a bank portal service by providing his ID and his password. The login information will first be validated by an authentication service where all customers information are stored. If the information is valid then the customer will be able to use the services of this bank. We focus in this case study on the credit request process only. Therefore, after logging in the customer will be able to request credit from the bank. The client can now place a new request, which includes the desired amount and the securities which form a balance. The balance is then assessed according to the output from the securities analysis service and the desired amount. If the balance is valid then the request is processed by the portal, which generates an automatic rating for the request; if not then the customer is asked to update his balance which will be assessed again.

According to the rating value one of the following scenarios will happen:

1. The client got "AAA" rating if the worth of securities is greater than the credit amount. This means that the client has a positive balance and the portal will generate an automatic offer for him, then the customer can accept or reject the offer.

2. The client got "BBB" rating if the worth of securities is at least half of the credit amount, then the client has a positive balance. In this case, the portal places the request with the rating in the task list of clerks for evaluation. Later, a clerk will retrieve a task from the list and evaluate it. If the risk is "HIGH" then the request is refused, otherwise a bank manager is consulted to approve the credit by placing the request, the rating and the clerk's evaluation in managers' task list. Then, a manager will retrieve the task and evaluate it. If the manager's decision is to approve the credit then an offer is generated, and the customer can accept or reject the offer. If the manager consulted refuses to accept the request then the client is notified accordingly.

3. The client got "CCC" rating if the worth of securities is less than the half of the credit amount. In this case the request is automatically rejected.

If an offer is generated and the client's decision is to accept the offer, then this offer is placed in the task list of an automated service which is notified to schedule the money transfer. Then the automated service will inform the client of the date of the transfer.

The client is able to cancel the request at any time in the process. If the client chooses to cancel the request then the request should be deleted from any task lists in which it exists.

### 7.3.2 Implementing the Finance Case Study in soaCSP

From the case study description we can conclude that the main services in our case study are: Client, BankPortal, CreditRequest, ClerkList, and ManagerList. Additionally, there are auxiliary services which are: Authentication, Rating, BalanceValidation, SecAnlaysis, and MoneyTransfer.

To design this scenario we start with the client establishing a session with the BankPortal service to access the bank services. If the client chooses to request a credit then the client service will establish a multiparty private subsession inside the bank to approve the client request. The details of the services are as follow:

The client first logs into the bank portal by providing his credentials (userID and password). More specifically, the **client** establishes a session with **BankPortal** then sends a message **login** containing the relevant information, then waits for either one of two messages: (**Valid**) if his credentials match an existing user, or (**notValid**) with an error message if not. If he receives the **Valid** message then he can start using the bank services. If the logging in failed then the client will be notified by the message **notValid.ErroMSG**, and the session will be closed. The client will then be asked to login again.

As stated in Section 7.3.1, we focus in this case study on the credit request process only. Therefore, after logging in the client will be able to request credit from the bank.

The client can request a credit by invoking the service **CreditRequest**, and asking it to join the existing session. The client then sends a message **request** containing the desired amount and the securities. This information forms the balance, which should be validated first by **CreditRequest** service, and the client will be notified accordingly about the results. If the balance is valid then the client receives the message (**validBalance**), and waits afterwards for either one of two actions that inform of the bank's decision: either he receives an offer which he can accept or reject, or he receives a message (**requestDenied**) where the session will be closed. In the case in which the request is approved, the message **transferDate** is sent informing of the date when the funds are to be made available. Alternatively, if the balance is not valid, the client is asked to enter the relevant information again.

$$
\begin{aligned}
&\texttt{client= BankPortal} \Leftarrow \{\ \texttt{login!ID!pass} \rightarrow \big(\ \texttt{(notValid.ErrorMSG} \rightarrow \texttt{SKIP)} \ \Box \\
&\qquad \texttt{(Valid} \rightarrow\ \ \texttt{CreditRequest} \Leftarrow^{+} \{\ \mu.\texttt{CRrec (request!amount!SEC} \rightarrow \\
&\texttt{(validBalance} \rightarrow \texttt{( userOffer} \ \Box\ \texttt{(requestDenied} \rightarrow \texttt{SKIP) ) )} \\
&\qquad \Box\ \texttt{(notvalidBalance} \rightarrow \texttt{CRrec) ) } \}\ \big)\ \} \\
&\texttt{userOffer= offer?amount} \rightarrow \texttt{( ( AcceptOffer} \rightarrow \texttt{transferDate?date} \rightarrow \texttt{SKIP)}
\end{aligned}
$$

```
⊓ ( RejectOffer → SKIP) )
```

Next we show the code of the **BankPortal** service. To indicate that the service is persistently available we use the notation $*$ with the service name. Firstly, the service receives the **login** message, then proceeds by opening a new subsession with the **Authentication** service to validate the client credentials. The **Authentication** service receives the request for validation via message **ValidID**. After that, the service will search the full clients database to match the user ID and password with an existing record. We assume that this database is a shared database outside the service code, therefore we use $\diamond$ to indicate that this communication is evaluated outside the border of this session. Both the **client** and **BankPortal** services will be notified accordingly via multi-synchronization communication about the result of searching the clients database. We use $\uparrow$ with these notification messages because they will be propagated to the upper session where the client service operates.

```
*BankPortal ⇒ login?ID?pass → Authentication ⇐ { ValidID!ID!pass
          → ( (notValid↑.ErrorMSG → SKIP) □ (Valid↑ → SKIP) ) }
*Authentication ⇒ ValidID?ID?pass → searchDB◇!ID!pass → getDBresult◇?ser
        → if (ser==exists) then (Valid↑ → SKIP)
        else (notValid↑.ErrorMSG → SKIP)
```

The **CreditRequest** service is the main service in this scenario. The service code starts by receiving the request, then proceeds by opening a new session with **BalanceValidation** service to validate the balance. Both the **CreditRequest** and the **client** services will be notified accordingly via multi-synchronization communication about the validation result. We use $\uparrow$ with these notification messages because they will be propagated to the upper session where the client service operates. If the balance is not valid then the client will be asked to re-enter his request with the right balance and then it will be validated again. If the balance is valid then the service **Rating** is asked to join the current session to calculate and provide a rating for this request. Afterwards, the service takes several branches according to the rating value as follows:

1. If the rating is "AAA", then an offer will be generated directly.
2. If the rating is "BBB", then the bank clerk will be asked to evaluate the risk of such request. The request with the rating will be added to clerks' task list, which is managed by service **Clerklist**. Later, the **Clerk** can retrieve the request and evaluate it, then reply with his evaluation via message **assessRisk** which carries the risk value and the evaluation.

Given the risk information, the **CreditRequest** service then acts according to the risk factor: if it is "HIGH" then the request is declined and the client is notified by message (**requestDenied**); otherwise the request with the rating value and the clerk's evaluation is placed in the mangers' task list, which is managed by service **Managerlist**, to seek a manager approval. Later, the **Manager** can retrieve the request and evaluate it, then reply with his decision via message **Mdecision.dec**.

After receiving the manager's decision, service **CreditRequest** acts accordingly: either an offer is generated denoting that the request has been approved, or the client is notified by means of a **requestDenied** message that the request has not been approved.

3. If the rating is "CCC", then the request is automatically declined and the client is notified by message **requestDenied**.

```
*CreditRequest ⇒ μ.Rrec ( request?amount?SEC  → BalanceValidation ⇐ {
       ValidateBC!amount!SEC →
      (validBalance↑ → Rating ⇐+ { getRate!amount!SEC  →
        returnRate?R → if R==AAA then generateOffer
          else if R=CCC then ( RequestDenied↑ → SKIP)
          else ProcessRequest } )  □ (notvalidBalance↑ → Rrec) )
ProcessRequest= ClerkList ⇐+ { addtoClist!amount!SEC!R →
      assessRisk?risk?eval → if risk==HIGH then (requestDenied↑ → SKIP)
          else ManagerList ⇐+ { addtoMlist!amount!SEC!R!eval →
                  Mdecision?dec → if dec==approve then generateOffer
                                  else (requestDenied↑ → SKIP) } }
```

The processes **generateOffer** and **ProcessRequest** are ordinary soaCSP processes which are used to organise the code of service **CreditRequest**. The code of the former process is provided below. The process of generating an offer starts by sending the offered amount to the client. If he accepts then service **TransferMoney** will be asked to join the current session to schedule the date of transferring the money. If he rejects the offer then the session is closed.

```
generateOffer= offer↑!amount → (( AcceptOffer↑ →
       TransferMoney ⇐+ { transMoney?amount → SKIP } )
                        □ ( RejectOffer↑ →  SKIP) )
*TransferMoney ⇒ transMoney!amount → transferDate↑!date → SKIP
```

As stated in the description of the **CreditRequest**, a new session is established with the **BalanceValidation** service to validate the balance. The code of **BalanceValidation** service starts by receiving the balance to be validated via message **ValidateBC**. To validate the balance, service **BalanceValidation** needs to analyse the securities. Therefore it invokes service **SecAnalysis** in the current session and sends securities to be analysed. According to the returned analysis result from the service **SecAnalysis**, the service **BalanceValidation** either sends the message **validBalance** to both the **CreditRequest** and the **client** services if the balance is acceptable (analysis is OK) or sends the message **notvalidBalance** to both the **CreditRequest** and the **client** services if the balance is not acceptable (analysis is notOK). We use $\uparrow$ with these notification messages because it will be propagated to the upper session where the client service operates.

```
*BalanceValidation ⇒ ValidateBC?amount?SEC  → SecAnalysis ⇐+ {
      analysis!SEC  → analysed?val → if val==OK then ( validBalance↑
      →  SKIP) else ( notvalidBalance↑ →  SKIP) }
*SecAnalysis ⇒ analysis?SEC  → analysisforSEC.SEC → analysed!val → SKIP
```

The **Rating** service is invoked by the **CreditRequest** service to give a rate to the current balance. The details of how the rate is calculated or how the securities are analysed are not a concern for this case study.

```
*Rating ⇒ getRate?amount?SEC  → calculaterate.amount.SEC →
                                              returnRate!R → SKIP
```

To implement the **ClerkList** and **ManagerList** services we use asynchronous communications. The stratagem here is to use the buffer attached to the asynchronous communication to act as the list. Moreover, the asynchronous communication is preferable here because the services need to interact with bank employees. The **ClerkList** service manages the clerks' list which is represented by channel **Clist**. The service code starts with the service **CreditRequest** sending the message **addtoClist** and sending the relevant information. The service then stores it in the buffer of the channel **Clist** by sending an asynchronous message via this channel with the relevant information, i.e, $Clist\diamond!<$. We use the symbol $\diamond$ to indicate that this list should be shared between all clients' requests, not only the request in the current session. However, to keep the state information of the current session (i.e., the session label), and to provide a way of communicating with the current session, we send with the request information a mobile channel (i.e., *thisuser*). Later, the clerk retrieves a request from the list, evaluates it, then temporarily merges with the

request session and returns the risk and the evaluation using the mobile channel *thisuser*. The sketch in Figure 7.1 graphically illustrates this service and its relevant processes.

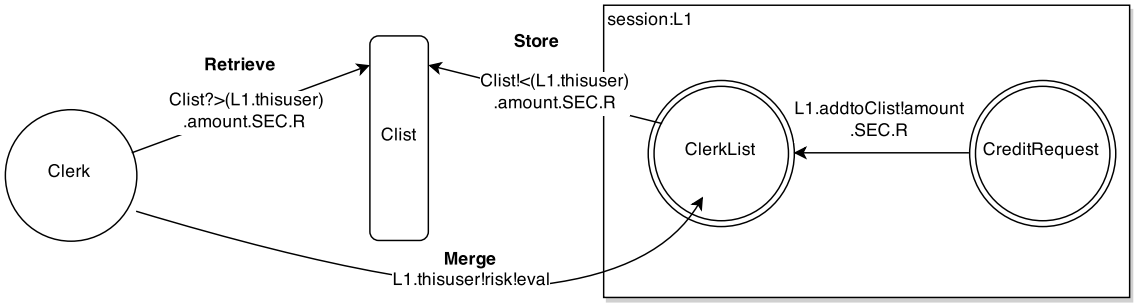


Figure 7.1: The finance case study messages sequence

The service **ClerkList** continues by sending back the risk value and the evaluation to the **CreditRequest** service. The **ManagerList** service works similarly to the **ClerkList** service. The difference is that it uses **Mlist** as the buffered channel and uses *thisEuser* to store the session state information. The details of how the clerks or managers evaluate the request are not a concern for this case study.

```
*ClerkList ⇒ addtoClist?amount?SEC?R → Clist◇!<(thisuser).amount.SEC.R →
    thisuser?>risk?eval → assessRisk!risk!eval → SKIP
*ManagerList ⇒ addtoMlist?amount?SEC?R?eval →
    Mlist◇!<(thisEuser).amount.SEC.R.eval → thisEuser?>dec →
     Mdecision!dec → SKIP
Clerk= Clist?>(x).amount.SEC.R → evaluateC  → x!<risk!eval  → Clerk
Manager= Mlist?>(x).amount.SEC.R.eval → evaluateM  → x!<dec  → Manager
```

We use the asynchronous mode to exchange information through the mobile channel *thisuser*, which simplifies the code and ensures the delivery of data. Alternatively, the **ClerkLists** service and **Clerks** process could synchronise on a channel known in advance, say *connect*. Then, through *connect*, **ClerkLists** sends the mobile channel with the (+) mark (i.e., *connect?(thisuser)*$^{+}$) in order to add it to the interface set which permits synchronisations through this mobile channel. The same thing applies to the mobile channel *thisEuser*.

The challenging behaviour in this case study is the **cancel** option of the client. If the client cancels his request then he should not receive any messages regarding this request

from the system. Additionally, the system should propagate the cancel order through the session to close the current session and delete the client request from lists if it has been added. To implement this behaviour we do the following:

First, we introduce the **cancel** event which represents the client cancel order. This event should be evaluated if the client wants to cancel his request. With this event and in order to capture the cancel option from the client interface, we install non-deterministic choices in all places where the system delivers information to the client. This ensures that if the client cancels his request then this order will resolve the choice and he will not receive any more messages regarding his request. The client process could be updated as follows:

```
client= BankPortal ⇐ { ( login!2!2 → ( (
       (notValid?ErrorMSG → SKIP)  □
        (Valid → ( CreditRequest ⇐+ { ( μ.CRrec (
          ( request!2!2 → ( ( ( validBalance →  (( userOffer
                                  □ (requestDenied → SKIP))
            ⊓ (cancel→SKIP)) )
         □ (notvalidBalance → CRrec) )
                          ⊓ (cancel → SKIP) ) )
                                    ⊓ (cancel→SKIP) ) ) }
              ⊓ (cancel→SKIP) ) ) )
           ⊓ (cancel→SKIP) )
        ⊓ (cancel→SKIP)) }

userOffer= offer?2 → ( ( ( AcceptOffer → ( (transferDate?date → SKIP)
           ⊓ (cancel→SKIP) )   ⊓
             ( RejectOffer → SKIP))) ⊓ (cancel→SKIP))
```

Secondly, in the case in which the cancel option is chosen by the client, then services should be informed that they should interrupt their execution, close the session, and remove the request from lists to which it has been added. This can be achieved by installing deterministic choices with the **cancel** event in the code of these services, in all the places where the system should deliver information to the client. These deterministic choices are used by services to check if a cancel order has been issued by the client before sending any data to him. The deterministic choices in the service code will be resolved by the non-deterministic choices in the client code.

If the cancel option has been announced in the session then the session should close

rather than sending this data, therefore we evaluate *SKIP* after the cancel event.

In addition to closing the session, any evaluation order that has been placed in the lists of Managers or Clerks should be deleted. In order to implement this behaviour we check for cancel order from the client before adding or retrieving data from these lists. If a cancel order is detected then we design a loop which deletes the client request from these lists.

The code of these services should be updated as follows:

```
*BankPortal ⇒ ( login?ID?pass → Authentication ⇐ {
                ( ValidID!ID!pass →( (
                (notValid↑?ErrorMSG → SKIP) □
                 (Valid↑ → SKIP) )  □ (cancel↑ →SKIP)) ) }
             □ (cancel↑ →SKIP))
*CreditRequest ⇒ ( μRrec.( ( request?amount?SEC →
             BalanceValidation ⇐ {  (  ValidateBC!amount!SEC →
             (( ( validBalance↑ → Rating ⇐+ {
             ( getRate!amount!SEC → returnRate?R →
                if (R == AAA) then generateOffer(amount)
           else if (R == CCC) then ((up(requestDenied) → SKIP)
             □ (cancel↑ →SKIP))
           else ProcessRequest(amount,SEC,R)   ) }
                  )  □ (notvalidBalance↑ → Rrec)
                           )  □ (cancel↑ →SKIP) )
                                )  }  )  □ (cancel→SKIP) ) )
generateOffer(amount)= offer↑!amount →( ( ( AcceptOffer↑ →(
               TransferMoney ⇐+ { transMoney!amount → SKIP }
             □ (cancel↑ →SKIP)) ) □ ((RejectOffer↑ →  SKIP)
               □ (cancel↑ →SKIP)) )  □ (cancel↑ →SKIP))
*Authentication ⇒ ValidID?ID?pass → searchDB⋄!ID!pass →
             getDBresult⋄.exists → if (exists == exists)
                           then ( (Valid↑ →  SKIP)  □ (cancel↑ →SKIP))
              else ((notValid↑!InvalidName → SKIP)  □ (cancel↑ →SKIP))
*TransferMoney ⇒ transMoney?amount → scheduletransfer
                 → ((transferDate↑!2 → SSKIP)  □ (cancel↑ → SKIP))
*BalanceValidation ⇒ ValidateBC?amount?SEC  → SecAnalysis ⇐+ {
            (analysis!SEC  → analysed?val → if (val == OK)
              then (( validBalance↑ →  SKIP)  □ ( cancel↑ →SKIP))
```

```
                else (( notvalidBalance↑ → SSKIP) □ ( cancel↑ →SKIP)) ) }
*SecAnalysis ⇒ analysis?SEC  → secanalysisE → analysed!OK → SKIP
*Rating ⇒ getRate?amount?SEC → calculaterate.amount.SEC →
              returnRate!BBB → SKIP
ProcessRequest(amount,SEC,R)= ClerkList ⇐+ { (( addtoClist!amount!SEC!R
               → assessRisk?risk?eval → if (risk == HIGH)
                 then ((requestDenied↑ → SSKIP)  □ (cancel↑ →SKIP))
                 else ManagerList ⇐+ {  ((addtoMlist!amount!SEC!R!eval
                    → Mdecision?dec → if (dec == approve)
                                      then generateOffer(amount)
                                      else (( requestDenied↑ → SKIP)
                     □ ( cancel↑ →SKIP)) )
                    □ ( cancel↑ →SKIP)) }
                          )  □ (cancel↑ →SKIP)) }
*ClerkList ⇒ addtoClist?amount?SEC?R → ( Clist◇!<(thisuser).amount.SEC.R
               → (  (thisuser◇?>?risk?eval → assessRisk!risk!eval → SKIP)
                □ ( cancel↑ → μdel.(Clist◇?>(x).amount.SEC.R →
               if (x == thisuser) then SKIP else (Clist◇!<(x).amount.SEC.R
                   → del)) ) ) □ ( cancel↑ →SKIP))
Clerk= Clist?>(x).amount.SEC.R → evaluateC  → x!<LOW!2  → Clerk
*ManagerList ⇒ addtoMlist?amount?SEC?R?eval → (
                Mlist◇!<(thisEuser).amount.SEC.R.eval → (
                ( thisEuser◇?>dec → Mdecision!dec → SKIP)  □
                 (cancel↑ → μmdel.(Mlist◇?>(x).amount.SEC.R.eval
                      →  if (x == thisEuser) then SKIP else
                              (Mlist◇!<(x).amount.SEC.R.eval → mdel)) ))
                                 □ (cancel↑ →SKIP))
Manager= Mlist?>(x).amount.SEC.R.eval →  evaluateM  → x!<approve → Manager
```

This implementation ensures that if the client cancels his request then the related session will be closed. However, the system will be frozen rather than terminating properly. This is because the non deterministic choice in the client code will resolve the deterministic choice in the current working service.

While we managed to stop the processing of this request, the system will not terminate and disappear from the environment. Using the *LISTEN* process will not suite our purpose here because it will close all sessions and terminate services if any termination is announced

anywhere in the system. Implementation of this behaviour should be easier if the compensation features of Chapter 8 are included in soaCSP. Having the compensations extension in soaCSP will terminate the system rather than freezing it. This addition is proposed as future work, and the result of this extension on the implementation of the case study is discussed in Section 8.5.

These services should collaborate in a system as follows:

$$
\begin{array}{l}
\texttt{system = client} \parallel_{\texttt{A}} \Big( \\
\quad \texttt{(*BankPortal} \parallel_{\{\mid\ ValidID,\ notValid\uparrow\ ,\ Valid\uparrow\ \mid\}} \texttt{*Authentication) |||} \\
\quad\quad \Big( \texttt{*CreditRequest} \parallel_{\texttt{B}} \\
\quad\quad\quad \big( \texttt{(*BalanceValidation} \parallel_{\{\mid \texttt{analysis,analysed} \mid\}} \texttt{*SecAnalysis) |||} \\
\quad\quad\quad\quad \texttt{*Rating ||| *TransferMoney ||| *ClerkList ||| *ManagerList } \big) \Big) \Big)
\end{array}
$$

```
Where
A={| login, Valid, notValid, request, validBalance, requestDenied, cancel,
       notvalidBalance, offer, AcceptOffer, RejectOffer, transferDate |}
B={| ValidateBC, validBalance↑ , notvalidBalance↑ , getRate,returnRate,
        transMoney ,addtoClist,assessRisk, addtoMlist,Mdecision, cancel |}
```

$$\texttt{finance = (system ||| (Clerks ||| Managers))} \parallel_{\{\mid\ a\leftarrow, a\rightarrow\ \mid\ a\in\ \Sigma\ \mid\}} \texttt{B}_{\Sigma}$$

The message sequence chart for the services' interactions is shown in Figure 7.3, and an illustrated sketch for this case study session hierarchy is provided in Figure 7.2.

### 7.3.3 Evaluating the case study in CSP tools

This case study shows an interesting scenario where not only multiple services interact in a session, but also the number of services that interact simultaneously in the session depends on some runtime conditions, e.g. the Clerk and Manager services are only invoked if the rating is "BBB", otherwise they do not participate in the collaboration; and the TransferMoney service is only asked to join in the session when the client accepts the offer.

As mentioned in the Introduction chapter, the advantage of using formal methods to model SOC systems is being able to reason on the correctness of these systems. In the previous section (Section 7.3.2), we used soaCSP functions in $\text{CSP}_M$, which were presented

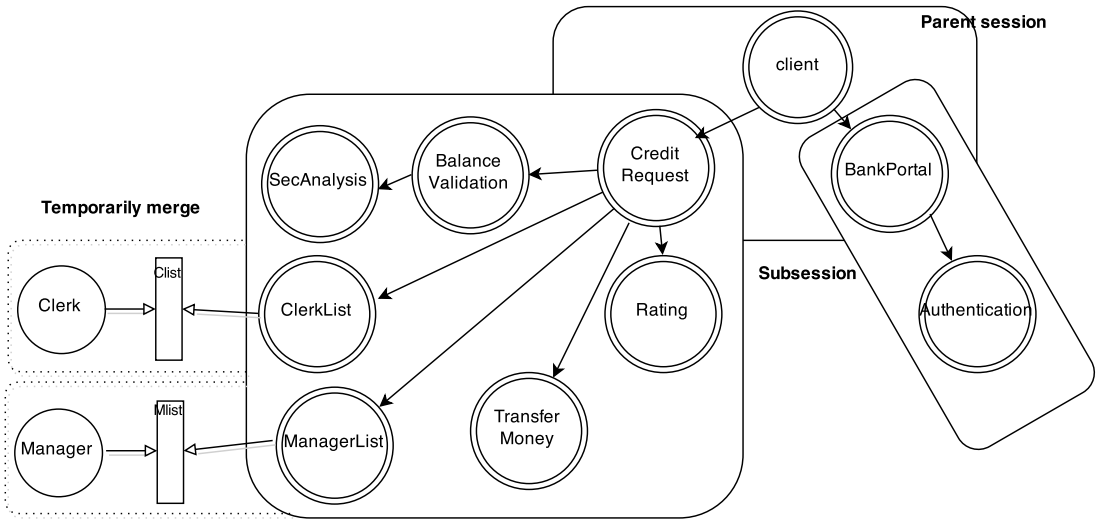


Figure 7.2: Session hierarchy of the finance case study

in sections 7.2.1, 7.2.2 and 7.2.3, to implement the finance case study[1]. In this section, we run this implementation in Probe [7], the trace animator of CSP, and FDR [12], the model checker of CSP. Firstly, we use Probe to demonstrate that our functions generate the right trace and work as desired. Secondly, we use FDR to prove some of the significant properties of the system.

The outputs of running the implementation of the finance case study in Probe are shown in the figures which are presented in this section.

Figure 7.4 shows the process of providing a unique label number for each service in the system, so 9 labels for 9 services. It shows also that the only possible trace for the client process in this system is to invoke BankPortal service. The invocation of the service BankPortal will open a new session. This session is labelled with the unique number which was given to the service by the system, i.e. 7 in this particular case.

In Figure 7.5, we demonstrate the scenario of logging in to the system. At point 1 we show that the cancel option can be captured before asking the user to login. At point 2 the trace shows that the client can login and then can still choose to cancel (exit the system). The logging scenario should be evaluated under the opened session $l.7$. To complete the logging in process the BankPortal process invokes the Authentication service in a new subsession $l.7.l.8$ as it appears in the trace and passes the information to it. The Authentication

[1]full code of soaCSP functions and the case study implementation is available at http://fac.ksu.edu.sa/sites/default/files/financecasestudy.pdf

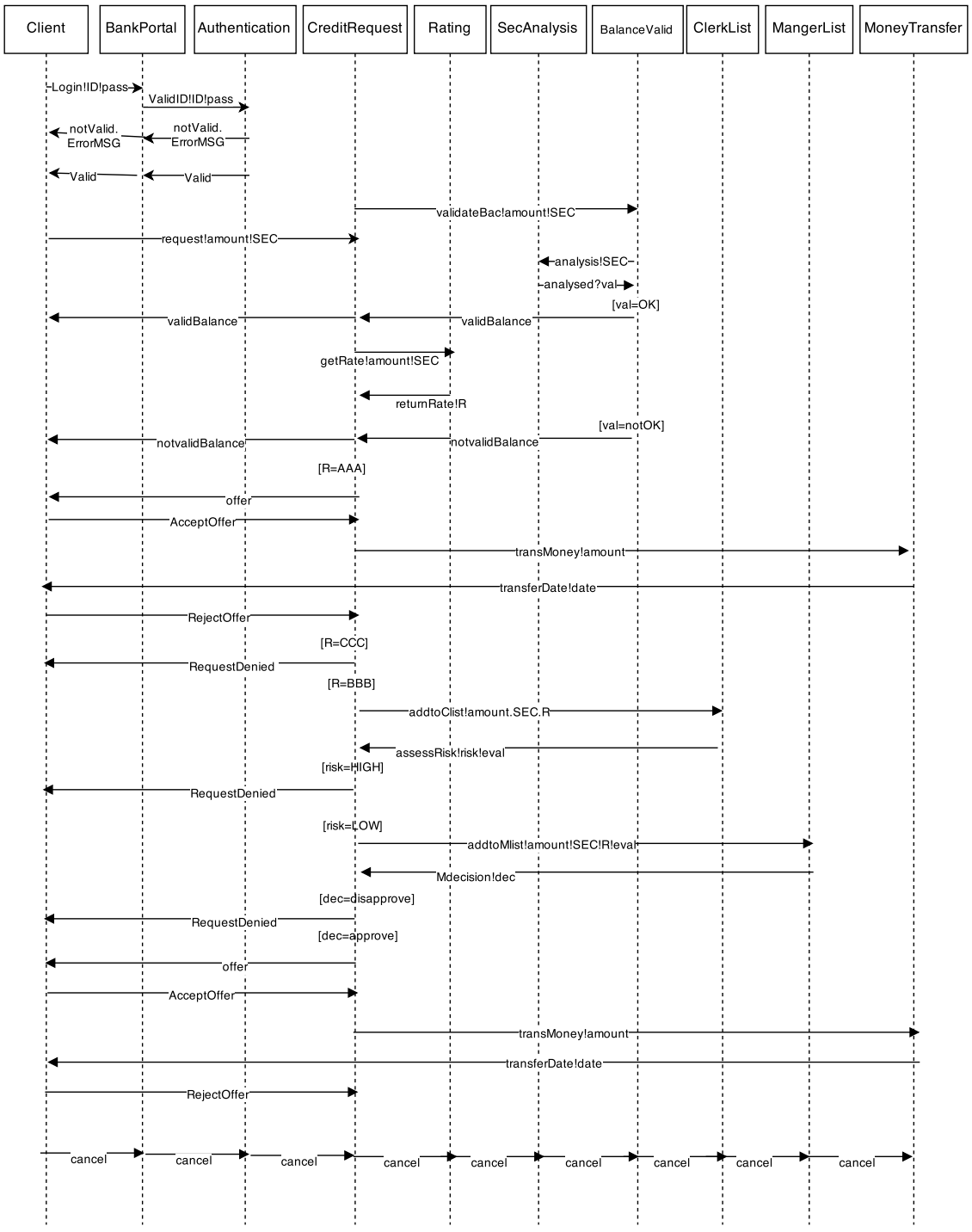


Figure 7.3: The finance case study messages sequence

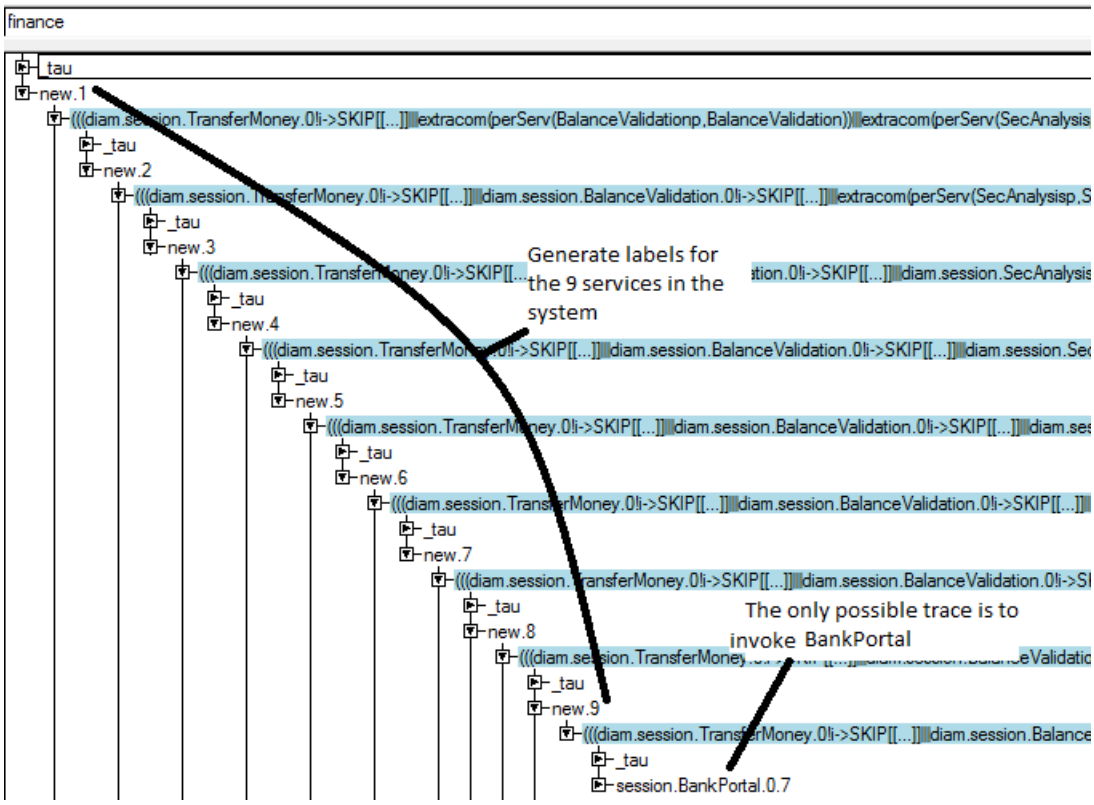


Figure 7.4: Issuing labels to services

service connects to the users' database to validate the user ID. The connection to the users' database is done outside the session's boundaries, i.e. unsessioned communications $\diamond$, so no labels are attached to this event. Although the client should be able to issue the cancel option here, the system will not respond because it is busy doing internal work.

In Figure 7.6, the trace shows, at point 1, that the Authentication service will send the validation result, Valid or notValid, simultaneously to the client and to the Bankportal service (we choose the Valid trace in this figure). Note that, the Authentication service sends the result to the parent session $l.7$, by using the upper communications $\uparrow$, although it works in the subsession $l.7.l.8$. The system can be interrupted before sending this result if the client cancels his request. It appears in the trace, at point 2, that the client can now invoke the CreditRequest service to request a credit. After the invocation the client can ask for loans by providing the relevant information. The system can also capture the cancel order from the client instead of evaluating the request order, as point 3 shows in the trace.

Following the process of issuing a request, Figure 7.7 presents the process of validating

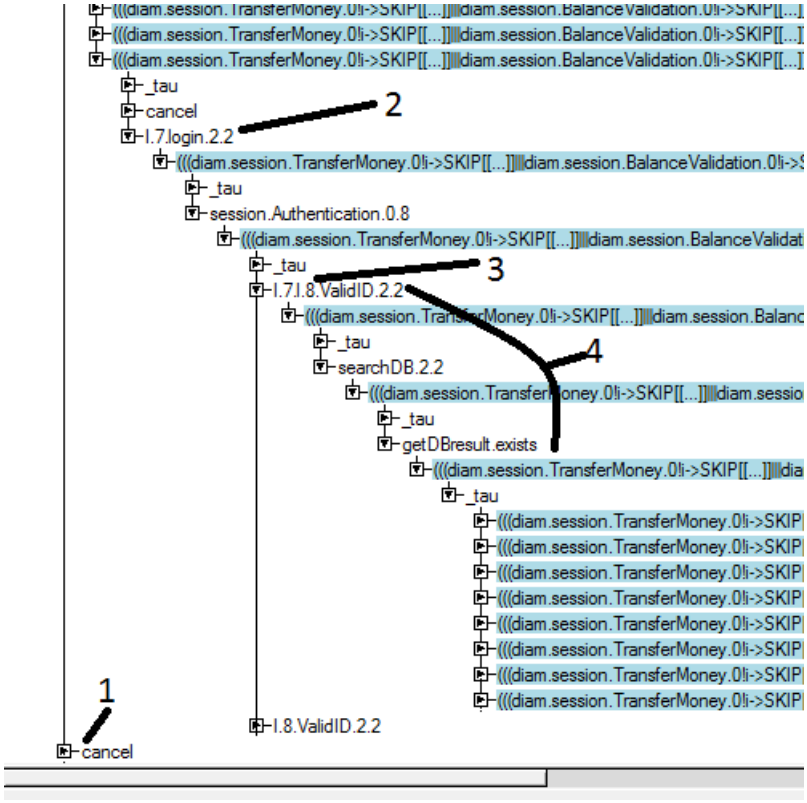


Figure 7.5: Logging in scenario

the provided balance. The process starts (see point 1 in the trace) by invoking the BalanceValidation service in a new subsession with the label shown in the figure $l.7.l.2$. The remainder of the trace presents the steps for validating a balance which involves invoking the SecAnalysis service to join the current session, therefore the label of this service (3 as shown in the figure) will be neglected and the current session label $l.7.l.2$ will be used instead, as can be seen at point 2 in the figure. The SecAnalysis service analyses the securities provided in the balance and sends the results back to the BalanceValidation service (here we choose the result to be OK).

As soon as the result of analysing the securities provided is received, the service BalanceValidation responds accordingly to both the client and the CreditRequest service by sending either the message validBalance or notvalidBalance in the same way as the message Valid was sent to them by the Authentication service. Following that, the CreditRequest service will respond accordingly by either asking the client to enter the correct data if the received message is notvalidBalance, or invoking the Rating service to join the current ses-

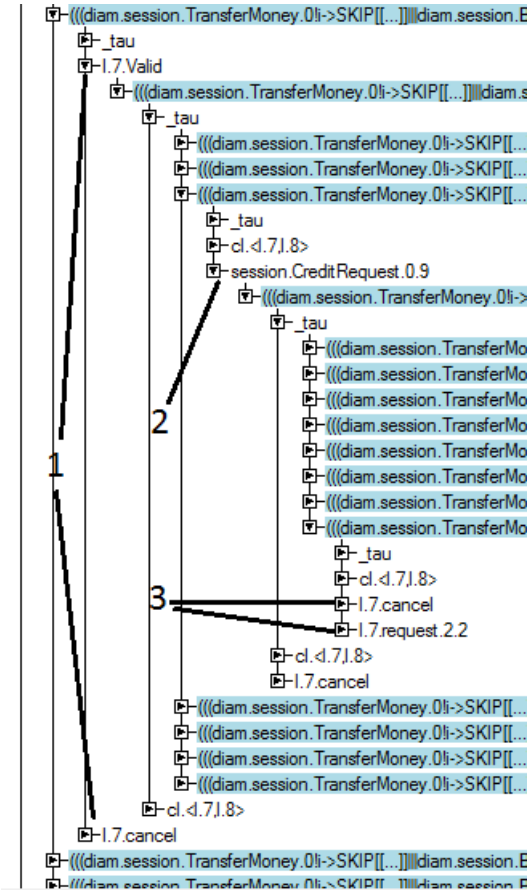


Figure 7.6: Issuing a request

sion to provide a rate for this request as shown in Figure 7.8 at point 1. According to the received rate, the CreditRequest service will respond as follows:

1. If the received rate is *CCC* then the CreditRequest service will automatically decline the request and send a message to the client informing him that his request was declined as shown in Figure 7.8 at point 2.

2. If the received rate is *AAA* then the CreditRequest service will automatically send an offer to the client as shown in Figure 7.9 at point 1.

3. If the received rate is *BBB* then the CreditRequest service places this request in the clerks' list for evaluation by invoking the ClerkList service to join the current session, then sending the request to it as shown in Figure 7.10 at point 1. Then

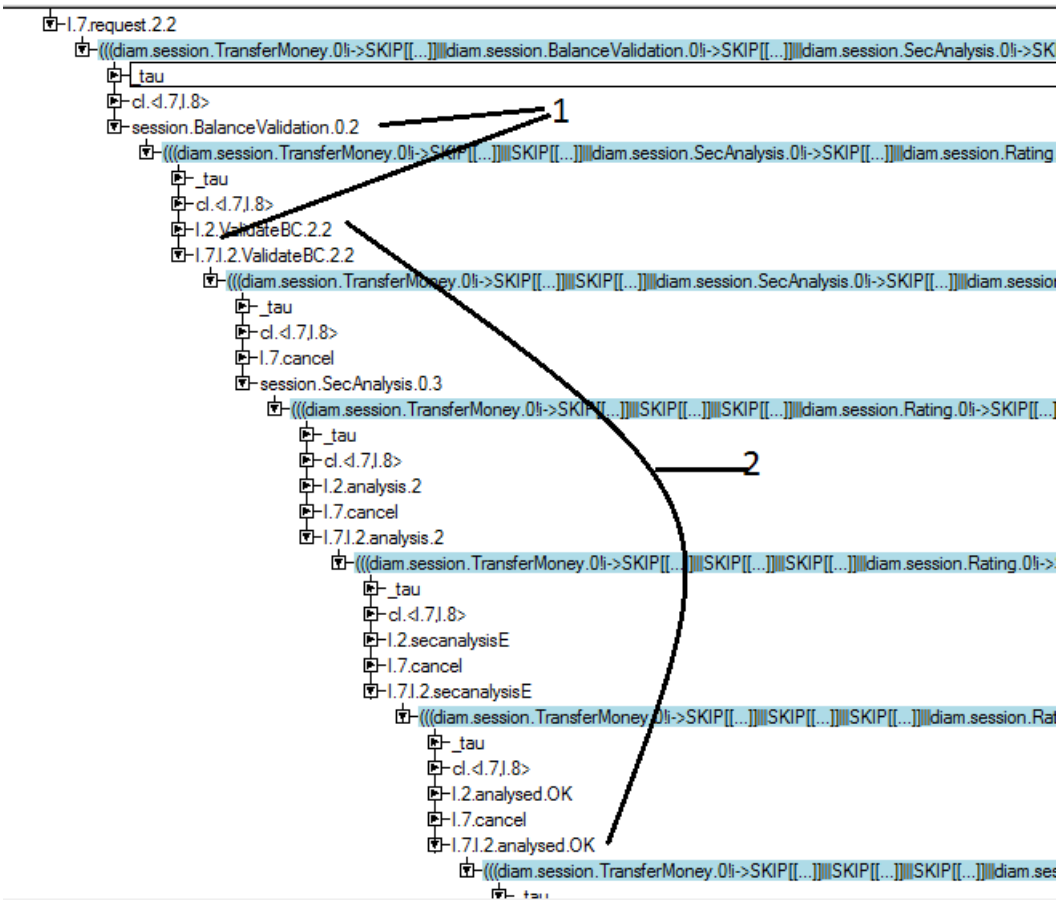


Figure 7.7: Validating a balance

the ClerkList places this request in the buffer of event Clist along with the mobile channel *thisuser* to save the state of this session and the relevant information, refer to point 2 in Figure 7.10. This buffer is automatically created in the system for any asynchronous communication which is accomplished in the system by using the keywords *in,out* with channel names as been explained in Section 7.2.1. Note that, all events are evaluated under session $l.7.l.2$, therefore the events have the label $l.7.1.2$. However, *Clist* event are unlabelled because it is unsessioned communication; refer to the implementation in Section 7.3.2 for details. *mthisuser* is the corresponding mobile alias for channel *thisuser*; refer to Section 7.2.2 for details.

If a clerk (presented in the system with process Clerks) is ready to evaluate this request, then he will retrieve this request from the buffer of the *Clist* event by eval-

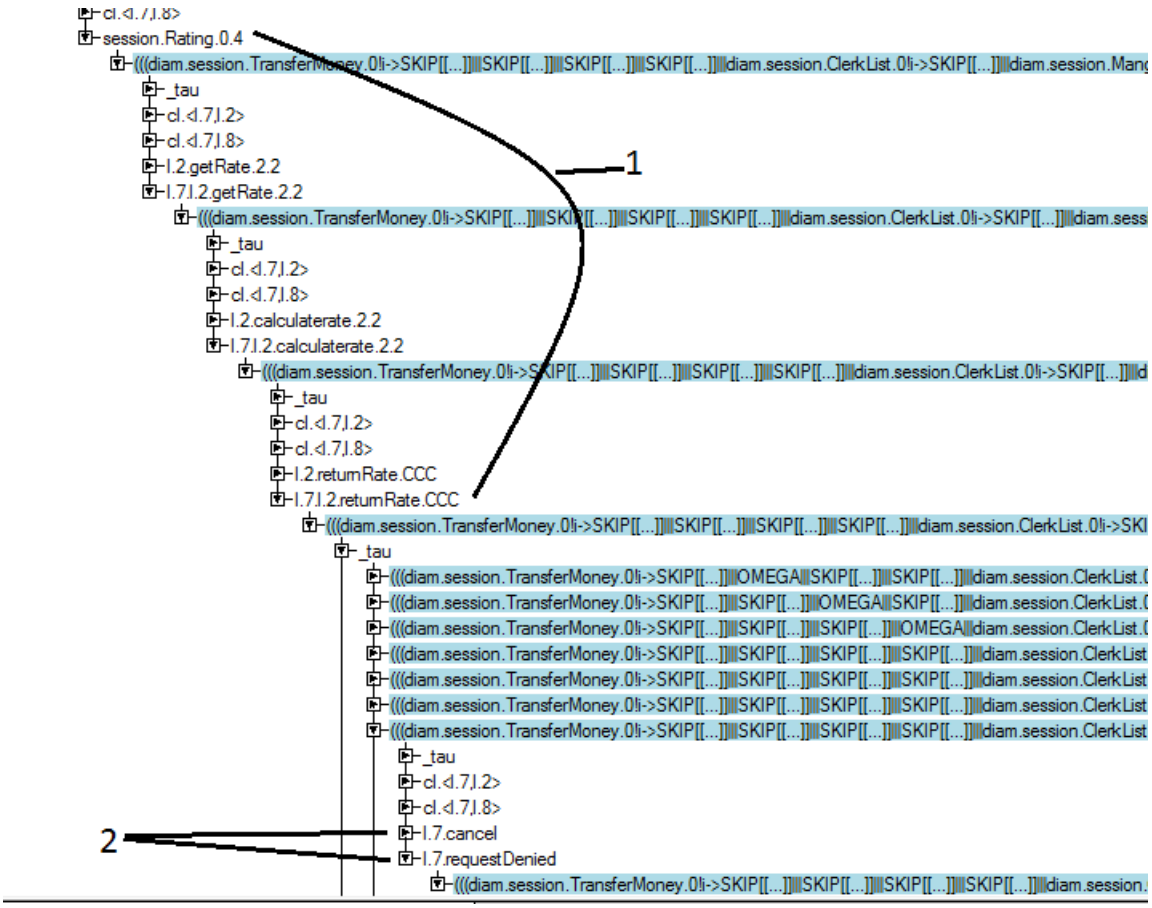


Figure 7.8: The rate *CCC* scenario

uating an asynchronous output communication via channel *Clist*, refer to point 1 in Figure 7.11. Moreover, refer to Figure 7.12 at point 1 to observe the data feeding and consuming in *Clist* buffer.

If a clerk has evaluated this request then the clerk will asynchronously send the evaluation via the mobile channel *thisuser* as shown at point 2 Figure 7.11. Later, the ClerkList service will asynchronously receive the evaluation and will then send it back to the CreditRequest service via channel *assessRisk*, see Figure 7.11 at point 4. If the received risk was *HIGH* then the CreditRequest service will reject this request automatically as was done with a request rated *CCC*. If the received risk was *LOW* then the CreditRequest service places this request in the managers' list for evaluation by invoking ManagerList service to join the current session, as shown in Figure 7.11 at point 3.

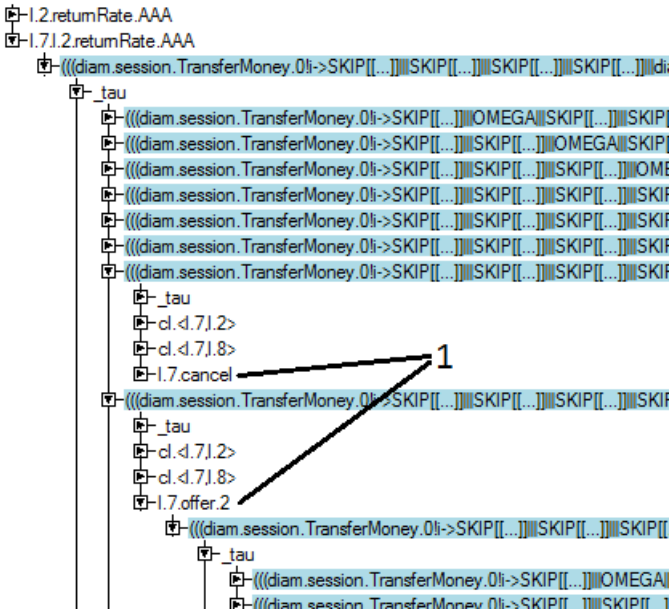


Figure 7.9: The rate $AAA$ scenario

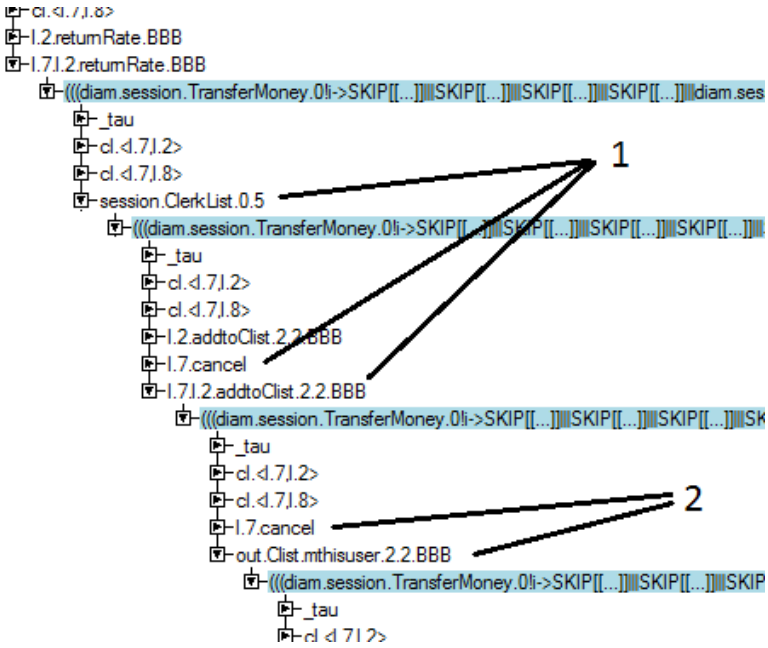


Figure 7.10: The rate $BBB$ scenario

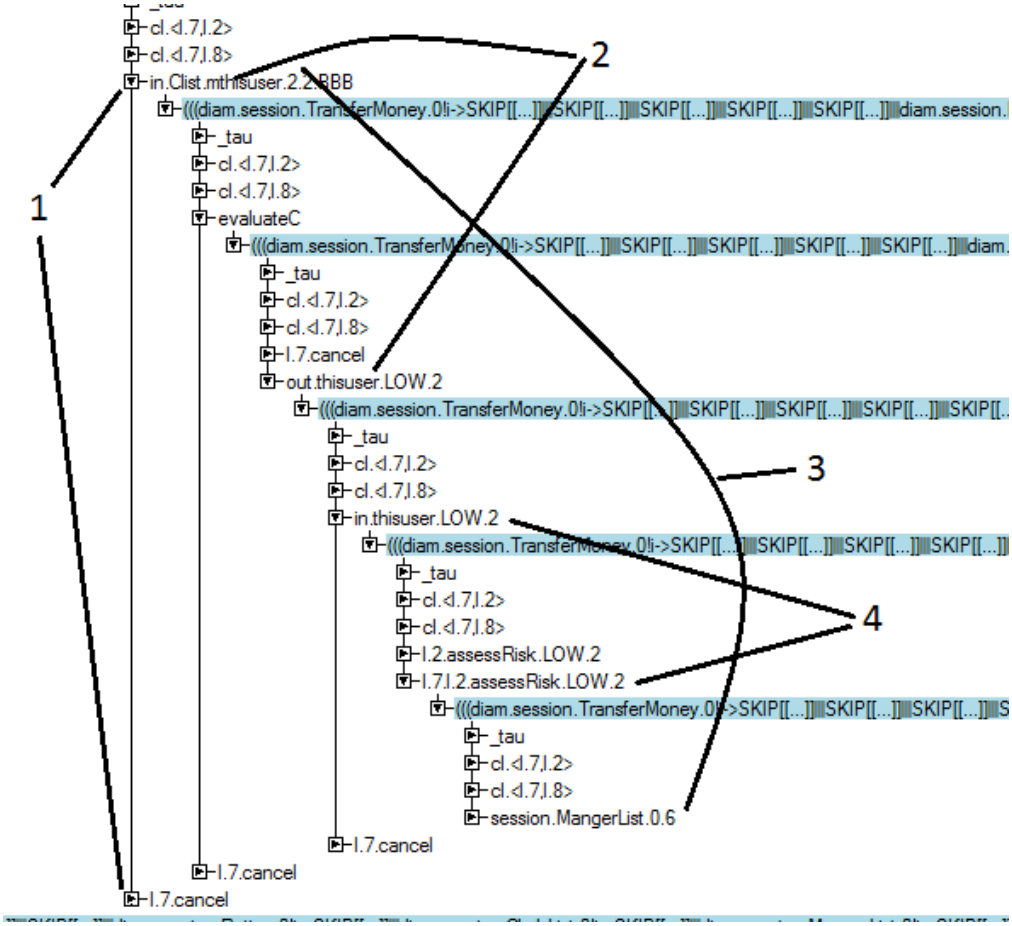


Figure 7.11: The processing of a request by the service ClerkList

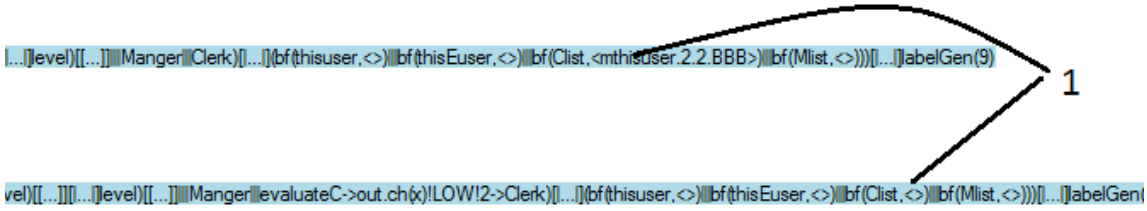


Figure 7.12: Data in buffers

The CreditRequest service sends the request to the ManagerList service which in turn places this request in the buffer of event Mlist along with the mobile channel *thisEuser*

and the relevant information. Later, the ManagerList service will asynchronously receive the manager's decision regarding this request, and will then send it back to the CreditRequest service via channel *Mdecision*, see Figure 7.13 at point 1. If the manager's decision is to approve the request then an offer will be generated for the client as was done with a request rated $AAA$.

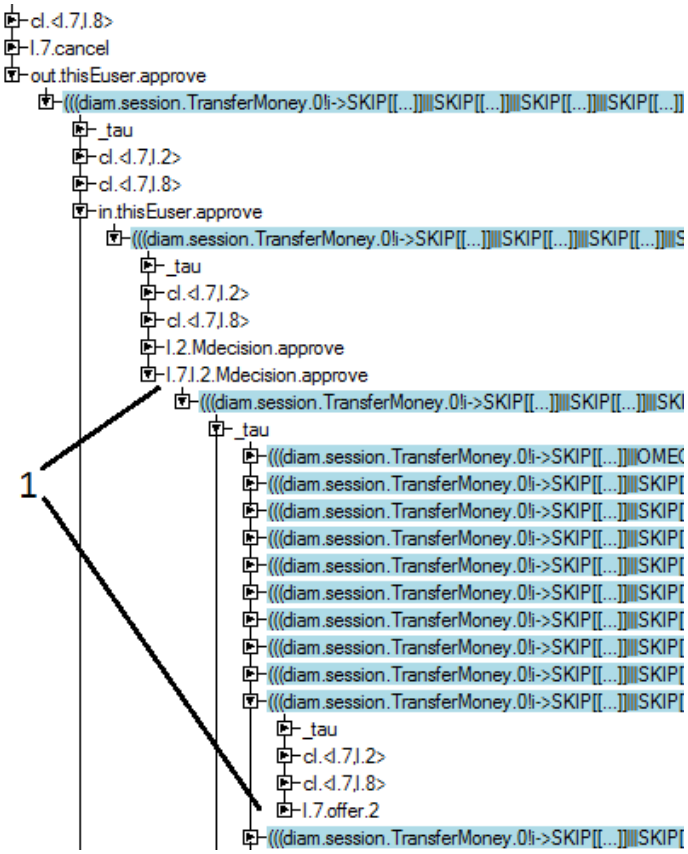


Figure 7.13: The processing of a request by the service Manager List

If an offer is generated for the client as been shown in Figure 7.14 point 1, then the client can accept the offer, reject the offer, or cancel the request, and the CreditRequest service will be informed accordingly via messages *AcceptOffer*, *RejectOffer*, or *cancel* as points 2 and 3 show in Figure 7.14. If the client accepts the offer then the CreditRequest service will invoke the TransferMoney service to join the current session, and pass the amount in the accepted offer message, to the service, see point 1 in Figure 7.15. If the client has not cancelled the request yet then the TransferMoney service will arrange directly with the

client a date to transfer the money to his account, as been shown at point 2 in Figure 7.15. Note that any messages directed to the client should be labelled with the label of the client session $l.7$. Therefore, the transferDate message is labelled with $l.7$ by using the upper communication $\uparrow$, instead of the current label of the TransferMoney service $l.7.1.2$.

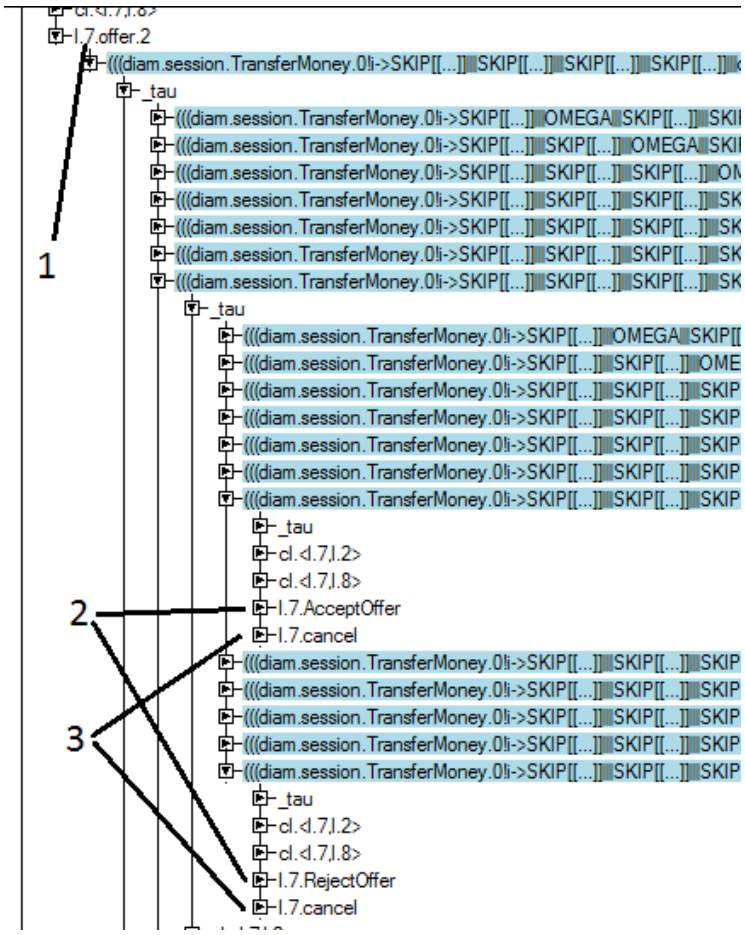


Figure 7.14: The processing of an offer by the client process

If at any point the user wants to cancel the offer, then the system should be interrupted and if the request has been placed in clerks' or managers' lists then it should be deleted. Figure 7.16, at point 1, shows the process of deleting a request from the clerks' list if cancel has been activated. The deletion is accomplished by retrieving the stored data in Clist. Point 1 in Figure 7.17 shows the other end of the trace where the evaluation of the deletion process is presented.

Finally,the termination behaviour of the system, as been explained in Section 7.2.3, is

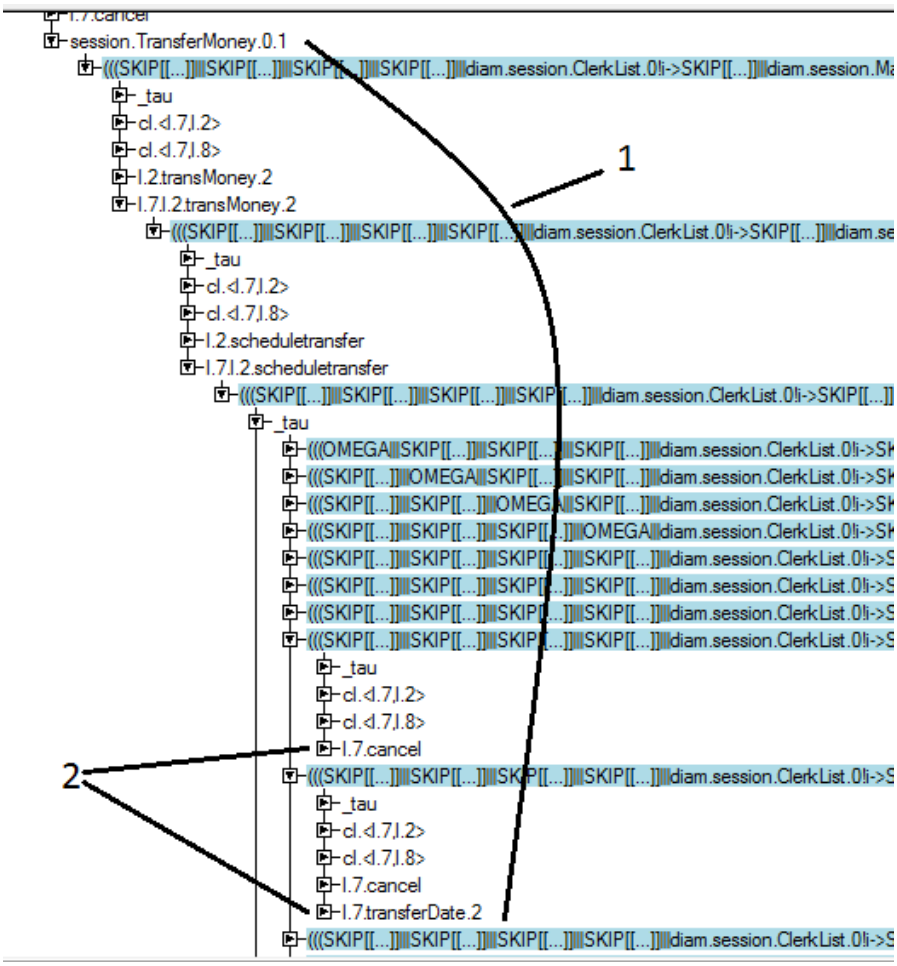


Figure 7.15: The processing of an accepted offer by the TransferMoney service

shown in Figure 7.18. Firstly, at point 1, event *cl* is storing the session labels of the sessions to be closed. Each session hierarchy has its own *cl* event. The event *cl* is evaluated within the process *SSKIP*. Secondly, at points 2 and 3, if *cl* is evaluated then a *close* event with the label of the session to be closed now is announced, and all sessions are synchronised on it. The $\tau$ shown at point 4, appears when all *close* events are evaluated and the session terminates, then the system just follows the closing behaviour of CSP.

Running our implementation in Probe shows that our implementation externally behaves as expected. The previous figures present the events that can be evaluated after each step in the finance process. This is useful for debugging systems and ensuring that a specific event or a specific trace is one of the outputs of the tested process. However,

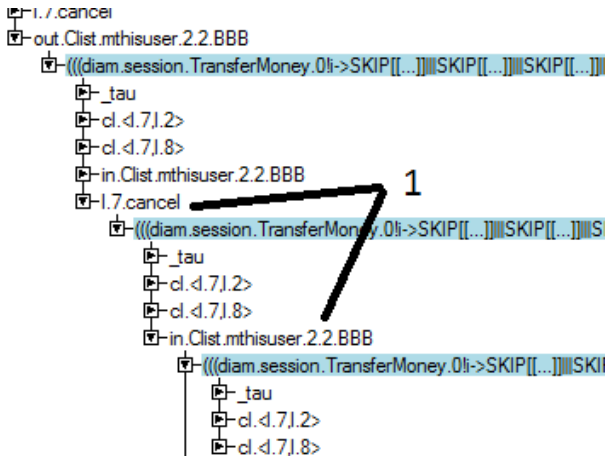


Figure 7.16: The processing of the cancel option

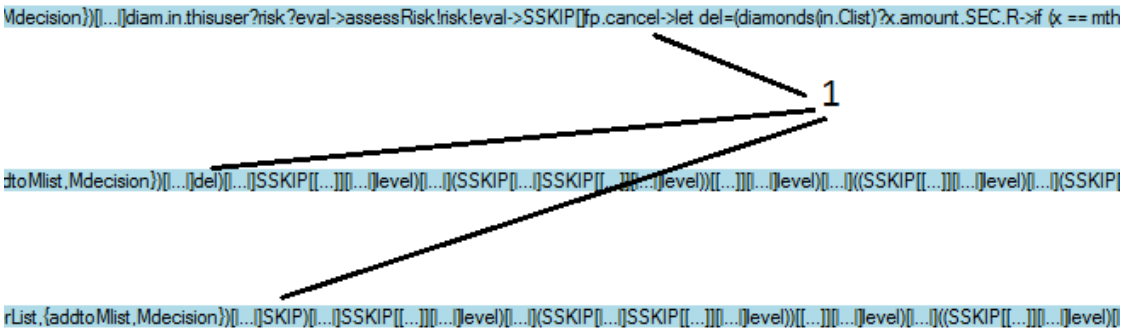


Figure 7.17: The deletion process for the request from Clist

enumerating traces does not prove any properties of the system. Therefore, we run our implementation in FDR [12], the Failure-Divergence Refinement model checker of CSP. In FDR, significant properties like deadlock-freedom, divergence-freedom, and determinism can be checked simply. Moreover, the trace, failure, and divergence equivalences permit reasoning on the equalities between processes in terms of their traces, failures, and divergences. In FDR, we can also check if a model refines another model in terms of their traces, failures, and divergences. These refinement checks are very useful for checking whether a process behaviour is contained within but not necessarily equal to another process's behaviour; for CSP equivalences and refinements details refer to Chapter 2 and [84, 124].

We run a version of our implementation without the cancel behaviour in FDR. This is

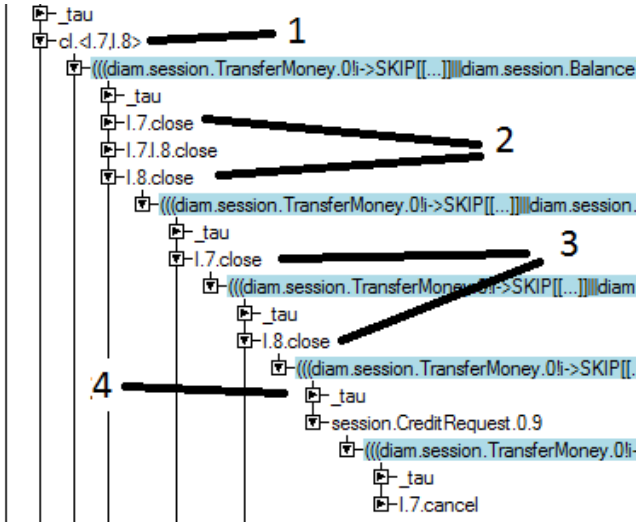


Figure 7.18: The termination behaviour of the system

because, the cancel behaviour will cause the system to deadlock, and it seems appropriate from the design point of view to delay the analysis of such behaviour until the content of Chapter 8 is incorporated into soaCSP.

The output of running the implementation of the finance case study in FDR is shown in Figure 7.19. In the FDR we checked the following properties:

- We proved that our implementation is deadlock free. This is because the assertion statement which asked FDR to check if the finance process is deadlock free passed with $\surd$. Therefore, the finance case study will never block.
- We proved that our implementation is livelock free. This is because the assertion statement which asked FDR to check if the finance process is livelock free passed with $\surd$. Therefore, the finance case study will never go into an endless loop.
- We checked if the finance case study is deterministic or not. This check failed as expected because we have in our implementation the non-deterministic choice in the client process, where the client non-deterministically can accept or reject offers.
- The fourth assertion in Figure 7.19 acknowledges Theorem 3.1. We proved that the set of traces produced by a buffered system (*asy(system)*) is contained in the unbuffered version of the system.

- In the last two assertions, we proved that the asynchronous and mobile communications introduce a delay in communication and provide a dynamic allocation for channels, but they do not change the behaviour of a system. This proof is achieved firstly by creating a process *financewithoutlists* which copies the behaviour of the finance process; however, instead of the services ClerkList and ManagerList adding the request to Clist and Mlist and then retrieving the evaluation through mobile channels, the services ClerkList and ManagerList in *financewithoutlists* do the evaluation internally. Secondly, we prove that the *financewithoutlists* process is trace equivalent to the *finance* process if we hide communications through *Clist*, *Mlist*, and the mobile channels in the behaviour of the *finance* process. Trace equivalence is proved by checking the trace refinement in both directions, i.e, *finance* refines *financewithoutlists* and *financewithoutlists* refines *finance*.

Unfortunately, properties like testing if the system will ever generate an offer for the client after his request was declined, cannot be checked by our implementation. Usually, this test can be checked by writing a trace like $requestDenied \rightarrow offer?amount \rightarrow SKIP$ then checking if this trace refines the finance process. If the assertion fails then this proves that this behaviour will never happen in the system. Although this property looks straightforward, it can not be evaluated by our implementation because in our implementation the event *requestDenied* is different from the event *l.7.requestDenied*. Therefore, this assertion will fail, but not for the right reason.

### General notes about soaCSP functions in CSP$_M$

- From the figures of the run of the finance case study in Probe, we can observe that if a subsession is created then the events can be evaluated according to either the full label or only the internal label of the session, e.g, event *transMoney* in Figure 7.15 can be evaluated under $l.7.l.2$ or $l.2$. This is because we implement the labelling algorithm of soaCSP as renaming relations. Therefore, two level of labelling are implemented by two renaming operators. However, the final result is the same, so this is not considered to be a mistake; see Section 7.2.3 for soaCSP labelling operator details.

- As Figure 7.18 shows, if the system terminates then the closure of both sessions will be available. However, this does not meet the requirements of the levelling up termination algorithm. We will investigate a solution to force the levelling up algorithm in future work; see Section 7.2.3 for the levelling up algorithm details.

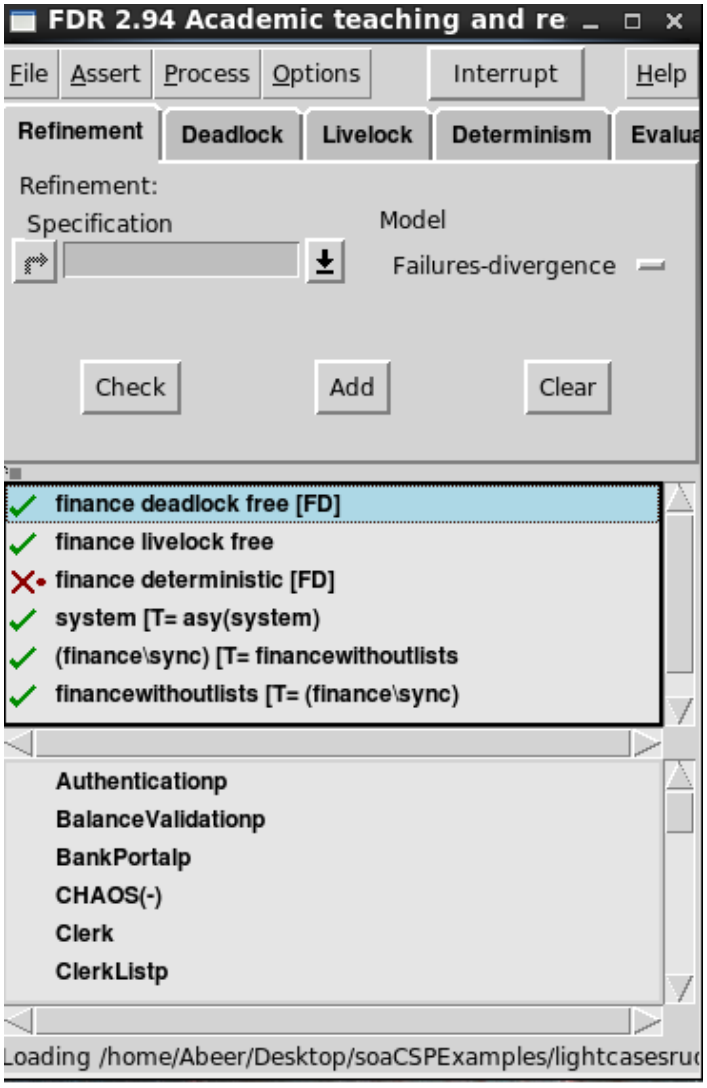


Figure 7.19: The finance case study in FDR

- As shown in Figure 7.12, in the code we use asynchronous unsessioned communication through the mobile channel *thisuser*. This is not compatible with our specification where we use asynchronous sessioned communication. This is because our implementation does not support the feature of passing labels along with mobile channels as we suggested in our theoretical model in Chapter 6. Adding this feature is proposed as future work.

## 7.4 Conclusions and Related Work

In this chapter, we present an implementation for soaCSP calculus in $\text{CSP}_M$. soaCSP was presented theoretically in Chapter 6 page 150, and $\text{CSP}_M$ is the machine readable version of CSP, and it is the input language for a range of CSP tools like Probe and FDR which were used in this chapter. The implementation successfully extends $\text{CSP}_M$ with functions to facilitate asynchronous, mobile, and session-based communications.

To the best of our knowledge, soaCSP is the first to provide automated checks through the model checker FDR for multiparty sessions and mobility. FDR checks include: trace, failure and divergence equivalence and refinements (see Chapter 2), deadlock-freedom, divergence-freedom, and determinism. This provides push-button proofs for these significant properties.

In the literature, several verification models have been designed like [64, 89, 71, 89, 112] to provide models for checking web services. However, as mentioned in the Introduction chapter, we are interested in models based on process calculi. This is because of the kind of reasoning mechanisms and verification properties that process calculi provide.

In the literature CMC [26] is proposed as a model checker for COWS calculus [100]. However, COWS as mentioned in Section 2.1 does not supports multiparty sessions.

Furthermore in the chapter, we demonstrated an implementation of our calculus for the finance case study [65, 66, 132] from the Sensoria project [8]. We remark that the produced code is concise compared to the description of the finance case study in BPEL [66] or in COWS [132]. This due to the fact that soaCSP supports multiparty sessions which reduce the number of services and messages which are used to circulate data.

Additionally, we found soaCSP helpful when modelling objects in the case study which were not necessarily a service such as bank employees. In the case study we model bank employees as ordinary processes and we permit these processes to communicate with the running session when needed.

Finally, we evaluate the implementation in Probe and in FDR to demonstrate the reasoning mechanisms of soaCSP.

# 8

# Compensating CSP (cCSP)

## 8.1 Introduction

Transactions are units of work comprised of a set of interactions between entities to achieve a final output [78]. The notion of a transaction is based on the idea of all-or-nothing, that is, none of the transaction's effects can take place until the whole transaction is committed.

Basically, there are two types of transactions: *Atomic Transaction (AT)* and *Long Running Transaction (LRT)* [78]. ATs prevent entities from updating system resources until the whole transaction is committed. Checkpoints and resources' key-locks are used to maintain systems in a safe state. LRTs relax the previous condition and allow the entities to update resources. However, LRTs use compensations to maintain systems in a safe state. Compensation is a technique to roll-back the system to a consistent state in the case of failure.

LRTs are intensively used in complex systems like SOC system where entities usually engage in transactions that last for hours, days or even longer while resources cannot be locked for such a long time. As a result, modelling languages for complex systems should be equipped with techniques to implement LRTs.

Considering process calculi as modelling languages, compensation has emerged as a crucial update. The fundamental idea behind compensating process calculi is to adapt the well-developed transaction techniques from database theory to the theory of process calculi, by introducing primitives to model and handle transactions. In essence, the key concepts introduced within process calculi are as follows:

- *Scope* which defines the transaction boundaries.

- *Fault* which represents an exception thrown by a process (internal fault); fault handlers are procedures which should be evaluated in such a case.

- *Termination* is the state of a process which is either committed (successfully terminated), or interrupted by other processes (external fault); termination handlers are procedures which should be evaluated in such a case. If nested transactions are allowed then the fate of these transactions should be determined in cases where the parent transaction terminates. In general there are three possibilities: sub-transactions will be aborted (i.e. terminated), will be discarded (i.e. deleted), or will be preserved (i.e. sub-transactions are levelled up and continue running).

- *compensation* is the reverse behaviour of a process, which should be evaluated in order to undo the effects of normal execution in the case of a failure; and this is to roll-back the system to a safe state. If compensations are considered then two issues should be carefully determined: firstly, the installation of a new compensation in a system, i.e. saving them until they are needed; secondly, the recovery mechanism which determines the evaluation order of the saved compensations, and the purpose of which is to recover the system to a consistent state after a failure.

In general, compensation can be statically implemented in any process calculus by creating a process, which captures a fixed compensation scenario that is planned in advance. Static compensations are feasible in systems where the evaluation of processes is fixed. However, in complex systems where there are interleaved, parallel and complex patterns of interactions, the compensation scenario heavily depends on the execution order. Therefore, compensating process calculi have been proposed as a suitable solution for modelling such complex systems.

The fundamental idea of such process calculi is introducing a new type of processes called **compensable processes**. A compensable process comprises two behaviours: the *forward behaviour* which corresponds to the normal execution of a process; and the *compensation behaviour* which corresponds to its reverse execution, which will be evaluated to undo the effects of the normal execution in case of a system failure. While a system is running and according to the execution order, these individual reverse behaviours are sorted and saved to incrementally and dynamically build a compensation scenario for this system.

Compensations have been introduced in a range of process calculi, including CCS [63], $\pi$-calculus [99, 135, 96], CSP [49] and Sagas [46, 44]. These compensating process calculi are either *interaction-based* or *flow-composition* calculi [43]. *Interaction-based* calculi associate

an explicit compensation sequence with each transaction, where new compensations can be added dynamically using new primitives, e.g. in [96], the construct $(inst \lfloor\rfloor)$ is used to update the transaction compensation sequence with new individual compensations.

On the other hand, *flow-composition* calculi do not use explicit compensation sequences. They dynamically construct one by composing, sequentially or in parallel, the individual compensations of processes which have successfully terminated. This is to undo the effects of these processes in the case of system failure.

In a case of system failure, i.e. a fault is announced in a transaction, the recovery mechanism of the calculus will evaluate the compensation sequence associated with this transaction. We distinguish between *static recovery*, which is the evaluation of a previously implemented compensation sequence, and *dynamic recovery*, which is the evaluation of a dynamically generated compensation sequence (i.e., generated while the system is running). In turn, dynamic recovery mechanisms can be classified as: *parallel recovery*, if all compensations are executed in parallel; *backward recovery*, if parallel processes are compensated in parallel and sequential processes are compensated in backward order; and *general dynamic recovery*, which is backward recovery with the option of replacing or discarding compensations at runtime [135, 95, 96].

Compensation has been introduced into CSP by Butler *et al* [49], who defined compensating CSP (cCSP) as a flow-composition calculus with a backward recovery mechanism. cCSP has been extended by Chen *et al* [57, 58] in the Extended compensating CSP (EcCSP), bringing back some significant operators from the original CSP and developing a theory of refinement.

In this chapter we extend cCSP further by introducing primitives to facilitate general dynamic recovery. We call the new calculus DEcCSP (Dynamic EcCSP). Improving the recovery mechanism from backward recovery to general dynamic recovery allows compensations to be replaced or discarded after they have been recorded. This is useful in many cases, such as when: (i) The compensation process is unknown at the start. (ii) The compensation process is subject to change while the process evolves. (iii) The compensation's logic is complex and spans several processes.

Moreover, DEcCSP extends EcCSP by including the remaining operators of the standard CSP, and that is to facilitate the specification of complex systems. These operators are: conditional (if-then-else), iteration (while-do), prefix operator, and named processes.

### Contributions of this chapter:

1. Formally introduce primitives to facilitate general dynamic recovery, where compen-

sations can be replaced or discarded after they have been recorded.

2. We formally extend EcCSP with the remaining operators of the standard CSP, and that is to facilitate the specification of complex systems. These operators are: conditional (if-then-else), iteration (while-do), prefix operator, and named processes.

**Structure of this chapter:** Section 8.2 recalls cCSP. Section 8.3 provides an operational semantics for EcCSP, which had so far only a denotational semantics. The operational semantics for EcCSP is used as a basis for the design, in section 8.4, of the syntax and the operational semantics of our calculus, DEcCSP. Section 8.5 illustrates its expressive power by updating the finance case study in Section 7.3 with compensations. Finally, Section 8.6 concludes the chapter and discusses related works.

## 8.2 Compensating CSP (cCSP)

In this section we recall the main concepts in cCSP [49], for details of CSP see Section 2.4.3 page 38.

The novelty in cCSP is the introduction of transaction processing features within the standard CSP processes. cCSP categorises processes into two types: standard processes, which are a subset of standard CSP processes, and compensable processes, which are standard processes attached to other standard processes to undo their effects. When a standard process terminates normally, it evolves to $\emptyset$, which means there is nothing to do; however, when a compensable process terminates normally its compensation will be preserved in case the transaction fails and the system needs to roll-back.

The syntax of cCSP is presented in Figure 8.1. cCSP includes operators for handling the key concepts of compensations presented in the introduction. We describe below the main constructs.

- *Scope*( [.] ), which identifies a transaction's boundaries.
- *Fault*, which is represented in cCSP with the new terminal signal (!). Fault can be explicitly introduced in a transaction using the $THROW$ primitive process. If a process throws then it terminates unsuccessfully. Fault handlers can be defined in cCSP using the $\triangleright$ operator, i.e., in $p \triangleright q$, $q$ is the fault handler of $p$.
- *Termination* in cCSP can be either successful termination using the standard CSP terminal signal $\surd$ (Recall that successful termination can be announced in a trans-

action by using $SKIP$ primitive process) or alternatively, a process can terminate if it is interrupted by another process. The interruption is presented in cCSP using the new terminal signal ?. A process can yield to an external exception and terminate by using the $YIELD$ primitive process. Transactions can be nested; sub-transactions should abort if the parent transaction terminates. In cCSP, transaction blocks cannot be interrupted.

- *Compensation* in a cCSP compensable process can be defined as a pair of standard processes which are composed with the new operator ($\div$), i.e. in $p \div q$, $q$ is the compensation handler of $p$.

In addition to the above constructs, the $\emptyset$ primitive process is added to the syntax of cCSP to represent a process which does nothing (similar to STOP in the original CSP). $a$ atomic process is the process which evaluates event $a$ then terminates successfully. In cCSP, processes can be composed sequentially or in parallel. Parallel composition are restricted to synchronise on terminal events solely. Choice between two processes is resolved by either of them performing an event.

The operational semantics of these operators and primitive processes is presented in Figure 8.2; it is based on the semantics presented by Ripon and Butler [50], but we have adapted it to follow the operational semantics of the standard CSP [121, 124].

In this chapter we use the following new notations: $pp, qq, \ldots$ denote compensable processes. $\Omega$ denotes the set of terminal events; $\omega, \omega'$ are used to range over this set. $\Sigma^{\tau\Omega}$ is the universal set $\Sigma^{\tau}$ union $\Omega$.

## 8.3 Extended cCSP (EcCSP)

Chen *et al* [57, 58] extended cCSP, adapted its trace semantics and developed stable-failure semantics and failure-divergence semantics for cCSP as in the standard CSP. They also brought back to the syntax of cCSP the original CSP operators: hiding, renaming, non-deterministic choice and recursion; see Chapter 2 for the details of these operators. Moreover, they changed the parallel composition operator to synchronise observable events, and introduced speculative choice ($\boxtimes$); a preliminary semantics for speculative choice was presented in Butler *et al* [49] and Ripon [120], but it was not included in the original cCSP. The EcCSP syntax is shown in Figure 8.3.

EcCSP [57, 58] was developed using denotational semantics. As previously mentioned in the Introduction chapter, we develop an operational semantics for our calculus to facilitate

(Standard Processes)

| | | |
|---|---|---|
| $p, q ::=$ | $a$ | (Atomic process) |
| | $p \Box q$ | (Choice operator) |
| | $p; q$ | (Sequential composition) |
| | $p \parallel q$ | (Parallel composition) |
| | $SKIP$ | (Primitive process) |
| | $THROW$ | (Primitive process) |
| | $YIELD$ | (Primitive process) |
| | $p \triangleright q$ | (Interrupt handler) |
| | $\emptyset$ | (Primitive process) |
| | $[pp]$ | (Transaction block) |

(Compensable Processes)

| | | |
|---|---|---|
| $pp, qq ::=$ | $p \div q$ | (Compensation pair) |
| | $pp \Box qq$ | (Choice operator) |
| | $pp; qq$ | (Sequential composition) |
| | $pp \parallel qq$ | (Parallel composition) |
| | $SKIPP$ | (Primitive process) |
| | $THROWW$ | (Primitive process) |
| | $YIELDD$ | (Primitive process) |

Figure 8.1: cCSP Syntax

the implementation in the model checker. To be consistent with the semantics presented in this thesis, and considering DEcCSP as an extension to EcCSP we need to develop an operational semantics for EcCSP. In this section, we develop an operational semantics for EcCSP based on the operational semantics of cCSP and CSP. Figures 8.4 and 8.5 present the new and the adaptive inference rules; the remainder of EcCSP is similar to cCSP. The adaptive rules are as follows:

- *Choice*, which can be deterministic or non-deterministic as in the standard CSP; see Chapter 2. Deterministic choice is resolved by either of the processes performing an observable or terminal event. On the other hand, non-deterministic choice is resolved by either of the processes performing the silent event "$\tau$".

- *Parallel composition*, which has been parameterised with an interface set as in the standard CSP. The purpose of this set is to govern the synchronisation between participants. Thereby, every event in this set should be performed simultaneously, otherwise events can interleave in any order.

- *Transaction block*, in EcCSP, the semantics of transaction block is adjusted to permit interruptions. In the same way atomic process semantics has been adjusted to permit

***Standard processes:***

Primitive processes: $\frac{}{SKIP \xrightarrow{\checkmark} \emptyset} \quad \frac{}{THROW \xrightarrow{!} \emptyset} \quad \frac{}{YIELD \xrightarrow{\omega} \emptyset} \quad (\omega \in \{?, \surd\})$

Process $a$: $\frac{}{a \xrightarrow{a} SKIP} \quad (a \in \Sigma)$

Choice: $\frac{p \xrightarrow{a} p'}{p \,\Box\, q \xrightarrow{a} p'} \quad \frac{q \xrightarrow{b} q'}{p \,\Box\, q \xrightarrow{b} q'} \quad (a, b \in \Sigma^{\omega\tau})$

Sequential Composition: $\frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q}\,(a \in \Sigma^{\tau}) \quad \frac{p \xrightarrow{\checkmark} p'}{p\,;q \xrightarrow{\tau} q} \quad \frac{p \xrightarrow{\omega} \emptyset}{p\,;q \xrightarrow{\omega} \emptyset}\,(\omega \in \{!, ?\})$

Interrupt Handler: $\frac{p \xrightarrow{a} p'}{p \rhd q \xrightarrow{a} p' \rhd q}\,(a \in \Sigma^{\tau}) \quad \frac{p \xrightarrow{!} p'}{p \rhd q \xrightarrow{\tau} q} \quad \frac{p \xrightarrow{\omega} \emptyset}{p \rhd q \xrightarrow{\omega} \emptyset}\,(\omega \in \{\surd, ?\})$

Parallel Composition: $\frac{p \xrightarrow{b} p'}{p \parallel q \xrightarrow{b} p' \parallel q} \quad \frac{q \xrightarrow{c} q'}{p \parallel q \xrightarrow{c} p \parallel q'} \quad (b, c \in \Sigma^{\tau}) \quad \frac{p \xrightarrow{\omega} \emptyset \quad q \xrightarrow{\omega'} \emptyset}{p \parallel q \xrightarrow{\omega \& \omega'} \emptyset} \quad (\omega, \omega' \in \Omega)$

*Where:*

| $\omega$ / $\omega'$ | $!$ | $?$ | $\surd$ |
|---|---|---|---|
| $!$ | $!$ | $!$ | $!$ |
| $?$ | $!$ | $?$ | $?$ |
| $\surd$ | $!$ | $?$ | $\surd$ |

Transaction block: $\frac{pp \xrightarrow{a} pp'}{[pp] \xrightarrow{a} [pp']} \quad \frac{pp \xrightarrow{!} p}{[pp] \xrightarrow{!} p} \quad \frac{pp \xrightarrow{\checkmark} p}{[pp] \xrightarrow{\checkmark} \emptyset}$

***Compensable processes:***

Compensation pair: $\frac{p \xrightarrow{a} p'}{p \div q \xrightarrow{a} p' \div q} \quad \frac{p \xrightarrow{\checkmark} \emptyset}{p \div q \xrightarrow{\checkmark} q} \quad \frac{p \xrightarrow{\omega} \emptyset}{p \div q \xrightarrow{\omega} SKIP}(\omega \in \{!, ?\})$

Primitive processes:

$SKIPP = SKIP \div SKIP \quad THROWW = THROW \div SKIP \quad YIELDD = YIELD \div SKIP$

$\frac{}{SKIPP \xrightarrow{\checkmark} SKIP} \quad \frac{}{THROWW \xrightarrow{!} SKIP} \quad \frac{}{YIELDD \xrightarrow{\omega} SKIP} \quad (\omega \in \{?, \surd\})$

Choice: $\frac{pp \xrightarrow{a} pp'}{pp \,\Box\, qq \xrightarrow{a} pp'} \quad \frac{qq \xrightarrow{b} qq'}{pp \,\Box\, qq \xrightarrow{b} qq'} \quad (a, b \in \Sigma^{\tau}) \quad \frac{pp \xrightarrow{\omega} p}{pp \,\Box\, qq \xrightarrow{\omega} p} \quad \frac{qq \xrightarrow{\omega'} q}{pp \,\Box\, qq \xrightarrow{\omega'} q} \quad (\omega, \omega' \in \Omega)$

Sequential Composition: $\frac{pp \xrightarrow{a} pp'}{pp\,;qq \xrightarrow{a} pp'\,;qq}\,(a \in \Sigma^{\tau}) \quad \frac{pp \xrightarrow{\checkmark} p}{pp\,;qq \xrightarrow{\tau} \langle qq, p\rangle} \quad \frac{pp \xrightarrow{\omega} p}{pp\,;qq \xrightarrow{\omega} p}\,(\omega \in \{!, ?\})$

*The auxiliary operator:* $\frac{qq \xrightarrow{a} qq'}{\langle qq, p\rangle \xrightarrow{a} \langle qq', p\rangle}(a \in \Sigma^{\tau}) \quad \frac{qq \xrightarrow{\omega} q}{\langle qq, p\rangle \xrightarrow{\omega} q\,;p}(\omega \in \Omega)$

Parallel Composition: $\frac{pp \xrightarrow{b} pp'}{pp \parallel qq \xrightarrow{b} pp' \parallel qq} \quad \frac{qq \xrightarrow{c} qq'}{pp \parallel qq \xrightarrow{c} pp \parallel qq'} \quad (b, c \in \Sigma^{\tau})$

$\frac{pp \xrightarrow{\omega} p \quad qq \xrightarrow{\omega'} q}{pp \parallel qq \xrightarrow{\omega \& \omega'} p \parallel q} \quad (\omega, \omega' \in \Omega)$

Figure 8.2: cCSP operational semantics

interruptions before or after performing its event.

The new operators introduced in EcCSP are: the *Hiding* operator, the *Renaming* operator, and the fixed point form of the *Recursion* operator. These operators are similar to the *Hiding*, the *Renaming*, and the *Recursion* operators of CSP. Additionally, EcCSP formally introduces the *Speculative choice* operator. The *Speculative choice* operator is the choice between two processes run in parallel without synchronisation. The choice is resolved

(Standard Processes)

| | | |
|---|---|---|
| $p, q ::=$ | $\ldots$ | (cCSP syntax) |
| $\mid$ | $p \sqcap q$ | (Internal choice operator) |
| $\mid$ | $p \underset{A}{\parallel} q$ | (Parallel composition) |
| $\mid$ | $p \backslash A$ | (Hiding operator) |
| $\mid$ | $p[\![R]\!]$ | (Renaming operator) |
| $\mid$ | $\mu p.f(p)$ | (Recursion) |

(Compensable Processes)

| | | |
|---|---|---|
| $pp, qq ::=$ | $\ldots$ | (cCSP syntax) |
| $\mid$ | $pp \sqcap qq$ | (Internal choice operator) |
| $\mid$ | $pp \underset{A}{\parallel} qq$ | (Parallel composition) |
| $\mid$ | $pp \backslash A$ | (Hiding operator) |
| $\mid$ | $pp[\![R]\!]$ | (Renaming operator) |
| $\mid$ | $\mu pp.ff(pp)$ | (Recursion) |
| $\mid$ | $pp \boxtimes qq$ | (Speculative choice operator) |

Figure 8.3: EcCSP Syntax

when one of these processes commits; consequently, the other process should immediately compensate. If both of them fail then the whole choice will fail.

## 8.4 Dynamic Extended cCSP (DEcCSP)

Improving the compensation recovery mechanism from backward recovery to general dynamic recovery allows compensations not only to be recorded in the right order dynamically, but to be discarded or replaced dynamically too. In this section, we endow EcCSP with primitives to facilitate general dynamic recovery. The main idea is to use a free process variable instead of the reverse behaviour process in compensable processes. The variable works as a place holder within the recovery sequence, where the real content can be retrieved later at the start of the execution. This will give the designer the ability to replace variable values whenever needed or discard them if they are no longer needed as long as the compensation has not been evaluated yet. Compensations can be discarded by assigning $SKIP$ to the process variable; $SKIP$ process is considered as an empty compensation.

The use of process variables to update compensations is inspired by the work of Guidi *et al* [79] in modelling the fault handler for SOCK [80] calculus. This idea has also been applied to the $\pi$-calculus in [95, 96]. Owing to the fact that these calculi are *interaction-based*, whereas DEcCSP is considered as a *flow-composition* calculus, we develop a different

**Adaptive rules:**

Process $a$:

*Completed:* $\frac{}{a \xrightarrow{a} SKIP}$ *Interrupted before event a:* $\frac{}{a \xrightarrow{?} STOP}$ *Interrupted after a:* $\frac{}{a \xrightarrow{a} STOP}$ $(a \in \Sigma)$

External (Deterministic) Choice: $\frac{p \xrightarrow{a} p'}{p \,\Box\, q \xrightarrow{a} p'}$ $\frac{q \xrightarrow{b} q'}{p \,\Box\, q \xrightarrow{b} q'}$ $(a, b \in \Sigma^{\omega})$

$\frac{p \xrightarrow{\tau} p'}{p \,\Box\, q \xrightarrow{\tau} p' \,\Box\, q}$ $\frac{q \xrightarrow{\tau} q'}{p \,\Box\, q \xrightarrow{\tau} p \,\Box\, q'}$

Parallel Composition: $\frac{p \xrightarrow{b} p'}{p \underset{A}{\parallel} q \xrightarrow{b} p' \underset{A}{\parallel} q}$ $\frac{q \xrightarrow{c} q'}{p \underset{A}{\parallel} q \xrightarrow{c} p \underset{A}{\parallel} q'}$ $(b, c \notin A)$ $\frac{p \xrightarrow{a} p' \quad q \xrightarrow{a} q'}{p \underset{A}{\parallel} q \xrightarrow{a} p' \underset{A}{\parallel} q'}$ $(a \in A)$

$\frac{p \xrightarrow{\omega} STOP \quad q \xrightarrow{\omega'} STOP}{p \underset{A}{\parallel} q \xrightarrow{\omega \& \omega'} STOP}$ $(\omega, \omega' \in \Omega)$

Transaction block: $\frac{pp \xrightarrow{a} pp'}{[pp] \xrightarrow{a} [pp']}$ $\frac{pp \xrightarrow{!} p}{[pp] \xrightarrow{!} p}$ $\frac{pp \xrightarrow{\surd} p}{[pp] \xrightarrow{\surd} STOP}$ $\frac{pp \xrightarrow{?} p}{[pp] \xrightarrow{?} p}$

**New rules:**

Internal (Non-deterministic) Choice: $\frac{}{p \sqcap q \xrightarrow{\tau} p}$ $\frac{}{p \sqcap q \xrightarrow{\tau} q}$

Recursion: $\frac{}{\mu p.f(p) \xrightarrow{\tau} f[\mu p.f(p)/p]}$

Hiding: $\frac{p \xrightarrow{b} p'}{p \setminus A \xrightarrow{b} p' \setminus A}$ $(b \notin A)$ $\frac{p \xrightarrow{a} p'}{p \setminus A \xrightarrow{\tau} p' \setminus A}$ $(a \in A)$ $\frac{p \xrightarrow{\omega} STOP}{p \setminus A \xrightarrow{\omega} STOP}$ $(\omega \in \Omega)$

Renaming: $\frac{p \xrightarrow{a} p'}{p[\![R]\!] \xrightarrow{b} p'[\![R]\!]}$ $(a \,R\, b)$ $\frac{p \xrightarrow{\tau} p'}{p[\![R]\!] \xrightarrow{\tau} p'[\![R]\!]}$ $\frac{p \xrightarrow{\omega} STOP}{p[\![R]\!] \xrightarrow{\omega} STOP}$ $(\omega \in \Omega)$

Figure 8.4: EcCSP standard processes operational semantics

method for updating and discarding compensations.

In addition to improving the recovery mechanism, we bring back the remaining operators of the standard CSP. These include: conditional (if-then-else) and iteration (while-do) control blocks, prefix operator, and named processes. Although control blocks can be simulated in CSP using the primitive operators as Hoare shows [84], Hoare also argues in [84] that in practice a reasonably wide range of operators is needed.

The rest of this section will be devoted to presenting DEcCSP's syntax and operational semantics in sections 8.4.1 and 8.4.2 respectively.

### 8.4.1 DEcCSP Syntax

The syntax of DEcCSP is given in Figure 8.6. DEcCSP extends EcCSP's syntax with operators to facilitate general dynamic recovery, as explained above. These operators include: (i) Assignment, to assign values to process variables. (ii)Variable compensation pair, where

**Adaptive rules:**

External (Deterministic) Choice: $\frac{pp \xrightarrow{a} pp'}{pp \,\Box\, qq \xrightarrow{a} pp'} \quad \frac{qq \xrightarrow{b} qq'}{pp \,\Box\, qq \xrightarrow{b} qq'} \quad (a, b \in \Sigma)$

$\frac{pp \xrightarrow{\omega} p}{pp \,\Box\, qq \xrightarrow{\omega} p} \quad \frac{qq \xrightarrow{\omega'} q}{pp \,\Box\, qq \xrightarrow{\omega'} q} \quad (\omega, \omega' \in \Omega) \quad \frac{pp \xrightarrow{\tau} pp'}{pp \,\Box\, qq \xrightarrow{\tau} pp' \,\Box\, qq} \quad \frac{qq \xrightarrow{\tau} qq'}{pp \,\Box\, qq \xrightarrow{\tau} pp \,\Box\, qq'}$

Parallel Composition: $\frac{pp \xrightarrow{b} pp'}{pp \underset{A}{\parallel} qq \xrightarrow{b} pp' \underset{A}{\parallel} qq} \quad \frac{qq \xrightarrow{c} qq'}{pp \underset{A}{\parallel} qq \xrightarrow{c} pp \underset{A}{\parallel} qq'} \quad (b, c \notin A)$

$\frac{pp \xrightarrow{a} pp' \quad qq \xrightarrow{a} qq'}{pp \underset{A}{\parallel} qq \xrightarrow{a} pp' \underset{A}{\parallel} qq'} \quad (a \in A) \quad \frac{pp \xrightarrow{\omega} p \quad qq \xrightarrow{\omega'} q}{pp \underset{A}{\parallel} qq \xrightarrow{\omega \& \omega'} p \underset{A}{\parallel} q} \quad (\omega, \omega' \in \Omega)$

**New rules:**

Internal (Non-deterministic) Choice: $\frac{}{pp \sqcap qq \xrightarrow{\tau} pp} \quad \frac{}{pp \sqcap qq \xrightarrow{\tau} qq}$

Recursion: $\frac{}{\mu pp.ff(pp) \xrightarrow{\tau} ff[\mu pp.ff(pp)/pp]}$

Hiding: $\frac{pp \xrightarrow{b} pp'}{pp \setminus A \xrightarrow{b} pp' \setminus A}\,(b \notin A) \quad \frac{pp \xrightarrow{a} pp'}{pp \setminus A \xrightarrow{\tau} pp' \setminus A}\,(a \in A) \quad \frac{pp \xrightarrow{\omega} p}{pp \setminus A \xrightarrow{\omega} p \setminus A}\,(\omega \in \Omega)$

Renaming: $\frac{pp \xrightarrow{a} pp'}{pp[\![R]\!] \xrightarrow{b} pp'[\![R]\!]}\,(a\,R\,b) \quad \frac{pp \xrightarrow{\tau} pp'}{pp[\![R]\!] \xrightarrow{\tau} pp'[\![R]\!]} \quad \frac{pp \xrightarrow{\omega} p}{p[\![R]\!] \xrightarrow{\omega} p[\![R]\!]}\,(\omega \in \Omega)$

Speculative Choice: $\frac{pp \xrightarrow{a} pp'}{pp \boxtimes qq \xrightarrow{a} pp' \boxtimes qq} \quad \frac{qq \xrightarrow{b} qq'}{pp \boxtimes qq \xrightarrow{b} pp \boxtimes qq'} \quad (a, b \in \Sigma)$

$\frac{pp \xrightarrow{\checkmark} p \quad qq \xrightarrow{\omega} q}{pp \boxtimes qq \xrightarrow{\checkmark} \langle q,p \rangle} \quad \frac{qq \xrightarrow{\checkmark} q \quad pp \xrightarrow{\omega} p}{pp \boxtimes qq \xrightarrow{\checkmark} \langle p,q \rangle}(\omega \in \{!, ?\})$

$\frac{pp \xrightarrow{\checkmark} p \quad qq \xrightarrow{\checkmark} q}{pp \boxtimes qq \xrightarrow{\checkmark} \langle q,p \rangle \,\Box\, \langle p,q \rangle} \quad \frac{pp \xrightarrow{\omega} p \quad qq \xrightarrow{\omega'} q}{pp \boxtimes qq \xrightarrow{\omega \& \omega'} \langle (\omega \& \omega'), (p \parallel q) \rangle} \quad (\omega, \omega' \in \{!, ?\})$

*The auxiliary operator:* $\frac{p \xrightarrow{a} p'}{\langle p,q \rangle \xrightarrow{a} \langle p',q \rangle}(a \in \Sigma^{\tau}) \quad \frac{p \xrightarrow{\omega} STOP}{\langle p,q \rangle \xrightarrow{\omega} q}(\omega \in \Omega)$

Figure 8.5: EcCSP compensable processes operational semantics

a process variable will take the place of the compensation in the usual compensation pair.

DEcCSP also includes the standard CSP operators: *if-then-else*, *while-do*, *prefix*, and named processes $N$, where process names can be used to specify recursive processes.

The traditional input/output notations $(?, !)$ coincide with the notations for terminal events $(?, !)$ introduced by Butler *et al* [49], however it will always be clear from the context which one is intended.

## 8.4.2 Operational Semantics of DEcCSP

We present below the operational semantics of DEcCSP, based on the operational semantics that we developed for EcCSP in Section 8.3. To keep track of the process variable values, we introduce a store to the semantics of DEcCSP. This store is a collection of process name locations which hold the current process names of the process variables in a model.

(Standard Processes)

| | | |
|---|---|---|
| $p, q ::=$ | $\ldots$ | (EcCSP syntax) |
| | $\mid$ if $b$ then $p$ else $q$ | (conditional control block) |
| | $\mid$ while $b$ do $p$ | (while control block) |
| | $\mid N$ | (Process name) |
| | $\mid a \longrightarrow p$ | (Prefix operator) |
| | $\mid X := p$ | (Variable assignment) |

(Compensable Processes)

| | | |
|---|---|---|
| $pp, qq ::=$ | $\ldots$ | (EcCSP syntax) |
| | $\mid$ if $b$ then $pp$ else $qq$ | (conditional control block) |
| | $\mid$ while $b$ do $pp$ | (while control block) |
| | $\mid NN$ | (Process name) |
| | $\mid p \div X$ | (Variable compensation pair) |

Figure 8.6: DEcCSP Syntax

More specifically, the configurations of the LTS of DEcCSP contain a global store denoted by $\rho$; we use $\rho$, $\rho', \ldots$ to represent its different states. We write $\rho(X)$ to denote the value of the process variable $X$ in $\rho$. $\rho[X \mapsto p]$ associates the process name $p$ with the variable $X$. Configurations are written $(p, \rho)$, or $(pp, \rho)$.

The global store keeps track of the values of process variables in the full space of configurations. Therefore, the state of the global store is only changed when a new process variable has been declared or if a process variable is assigned a new value.

Below we present the semantics of the new extensions in DEcCSP, the rest of DEcCSP is similar to EcCSP.

**General Dynamic recovery** can be implemented in the calculus by using a *compensation pair with process variable*. A compensation pair with process variable consists of a standard process as a forward behaviour and a process variable as its compensation partner. The variable works as a place holder within the recovery sequence, where the real content can be retrieved later.

$px$ will denote a compensation pair with process variable, to distinguish it from the standard one which is denoted by $pp$, i.e., $px = p \div X$, where $p$ is the standard forward behaviour of $px$, and the $X$ process variable is a place holder for the compensation behaviour of $px$. $X$ should be a fresh variable, i.e., not in the domain of the global store.

During the execution of the forward behaviour $p$, the value of $X$ can be changed anywhere in the system. If $p$ terminates normally then the variable $X$ will be recorded, and $X$ still can be changed anywhere in the system as long as the compensations have not been

evaluated yet. If $p$ terminates abnormally then so does the compensation pair, resulting in an empty compensation. This is formalised by the following rules.

$$\frac{(p,\rho) \xrightarrow{a} (p',\rho)}{(p \div X,\rho) \xrightarrow{a} (p' \div X,\rho)}\, a \in \Sigma \qquad \frac{(p,\rho) \xrightarrow{\surd} (STOP,\rho)}{(p \div X,\rho) \xrightarrow{\surd} (X,\rho)}$$

$$\frac{(p,\rho) \xrightarrow{\omega} (STOP,\rho)}{(p \div X,\rho) \xrightarrow{\omega} (SKIP,\rho)}\, \omega \in \{!,?\}$$

$X$'s value can be replaced by assigning a new value to it; to discard the compensation assign $SKIP$.

$$\frac{}{(X := p,\rho) \xrightarrow{\tau} (SKIP,\rho[X \mapsto p])}$$

The stored value of the process variable $X$ will be retrieved if the associated transaction throws an exception. If a transaction $([pp],\rho)$ throws an exception !, then the transaction block will be ended, and the corresponding compensation sequence $p$ will be evaluated. Recall that $p$ is the composed sequence of compensations of any successfully terminated processes within this transaction. At this time, the values of any process variable should be retrieved by replacing it with its value in the global store.

$$\frac{}{(X,\rho) \xrightarrow{\tau} (p,\rho)}\, \text{if } \rho(X) = p$$

**Control Blocks** *if-then-else* and *while-do* are the same as the standard control blocks, where $b$ in the two control blocks is a boolean expression which is evaluated according to the standard boolean semantics.

The If-then-else conditions in standard processes is defined as follows:

$$\frac{(b,\rho) \xrightarrow{\tau} (b',\rho)}{(if\ b\ then\ p\ else\ q,\rho) \xrightarrow{\tau} (if\ b'\ then\ p\ else\ q,\rho)}$$

$$\frac{}{(if\ True\ then\ p\ else\ q,\rho) \xrightarrow{\tau} (p,\rho)} \quad \frac{}{(if\ False\ then\ p\ else\ q,\rho) \xrightarrow{\tau} (q,\rho)}$$

The If-then-else condition in compensable processes is defined as follows:

$$\frac{(b,\rho) \xrightarrow{\tau} (b',\rho)}{(if\ b\ then\ pp\ else\ qq,\rho) \xrightarrow{\tau} (if\ b'\ then\ pp\ else\ qq,\rho)}$$

$$\frac{}{(if\ True\ then\ pp\ else\ qq,\rho) \xrightarrow{\tau} (pp,\rho)} \quad \frac{}{(if\ False\ then\ pp\, else\ qq,\rho) \xrightarrow{\tau} (qq,\rho)}$$

Iterations in standard and compensable processes are as follows:

$$\frac{}{(while\ b\ do\ p, \rho) \xrightarrow{\tau} (if\ b\ then\ (p; while\ b\ do\ p)\ else\ SKIP, \rho)}$$

$$\frac{}{(while\ b\ do\ pp, \rho) \xrightarrow{\tau} (if\ b\ then\ (pp; while\ b\ do\ pp)\ else\ SKIP, \rho)}$$

**Named Processes for Definitions and Recursion** We write $(N = p)$ if $N$ is the name of the standard process $p$, and $(NN = pp)$ if $NN$ is the name of the compensable process $pp$. Process names can be used in the more common style of recursion where the process name is used in the process body.

$$\frac{}{(N, \rho) \xrightarrow{\tau} (p, \rho)}\ (\text{if } N = p) \qquad \frac{}{(NN, \rho) \xrightarrow{\tau} (pp, \rho)}\ (\text{if } NN = pp)$$

***Prefix operator*** Let $a$ be an event in $\Sigma$, and let $p$ be a standard process, then $a \rightarrow p$ represents a standard process which is ready to engage in event $a$ and then behave as $p$.

$$\frac{}{(a \rightarrow p, \rho) \xrightarrow{a} (p, \rho)}$$

We remark that $a \longrightarrow p$ has the same expressive power as $a; p$ where $a$ here is an atomic process. Nevertheless, we extend EcCSP with the prefix operator for two reasons: firstly, to provide the standard method (in CSP) for writing processes; and secondly, to facilitate introducing compensations to soaCSP which is proposed as a future work (see Chapter 9).

## 8.5 Compensations in the Finance Case Study

Instead of using a simple example to illustrate the usability of DEcCSP we prefer to present, in this section, what would be the result if we combined DEcCSP with soaCSP, by showing how compensations can efficiently implement the **cancel** behaviour of the finance case study described in Section 7.3.1 page 184. This highlights the usability of DEcCSP in a real scenario. We assume that the compensation part of a compensable process will have the same label as the standard part. Additionally, we assume that the unsuccessful termination primitive processes $THROW$ and $YIELD$ will not be labelled at all. This is to allow the effects of these processes to propagate through the whole transaction even if the transaction contains services working in different levels of sessions.

As mentioned in Section 7.3.2 the **cancel** behaviour of the finance case study can be implemented efficiently by using DEcCSP extension; for the finance case study description refer to Section 7.3.1. Hence, the finance process with the **cancel** behaviour will terminate instead of freezing the request as we did in the implementation of the **cancel** option in soaCSP. Moreover, using compensations will provide efficient implementation for the client process as the **cancel** option can be directly evaluated.

Considering the implementation of the finance case study which was presented in Section 7.3.2, we adjust the implementation to encode the following behaviours:

1. The cancel option is evaluated as soon as the cancel option is chosen. Following that, the client process will throw an exception to force the whole system to terminate.

2. If the client process throws an exception then the active services in the system should yield to this exception by interrupting their execution and terminating.

3. If the services are interrupted after the client request has been placed in the clerks' list or the managers' list, then this request should be deleted from these lists.

The adjusted implementation of the finance case study should be as follows:

First, we should change the processes from normal processes to compensable processes. We can achieve that by converting events to atomic processes. Following that, we change these normal atomic processes to compensable processes by attaching compensations to them. For most of the events we will attach the empty compensation ($SKIP$) as their actions do not need to be rolled back. The two actions that should be rolled back are the addition of the client request to either clerks' or managers' lists. If the request has been added to these lists for evaluation then the evaluation request should be deleted if the client request has been cancelled. Finally, in each process or service we insert a $YIELD$ process to interrupt the execution of this process if there is a failure announced in the transaction. The $YIELD$ process is considered as a replacement of the choice ($\Box(cancel \longrightarrow SKIP)$) in the previous implementation of this case study in Chapter 7. We insert $YIELD$ in the places where this choice should be resolved, the same as in the previous version of this implementation.

To clarify this point see the adjusted code for the **Clerklists** service. Recall that, $YIELDD$ is the compensable version of the $YIELD$ primitive process, and the $SKIPP$ is the compensable version of the $SKIP$ primitive process.

```
*ClerkList ⇒  (addtoClist?amount?SEC?R ÷ SKIP) ; YIELDD;
```

```
                                (Clist◇!<(thisuser).amount.SEC.R ÷ deleteC) ;
                                  YIELDD; (thisuser◇?>risk?eval ÷ SKIPP) ;
                                       (assessRisk!risk!eval ÷ SKIP) ;  SKIPP

deleteC= μdel.( Clist◇?>(x).amount.SEC.R →
               if (x == thisuser) then SKIP else
                       ( Clist◇!<(x).amount.SEC.R → del ))
```

The implementation of the services BankPortal, CreditRequest, ManagerList, Authentication, Rating, BalanceValidation, SecAnlaysis, and MoneyTransfer can be adjusted in the same way as the implementation of the ClerkList service. The code of the client process can be adjusted in the same way as well; however, in addition to these adjustments the code of the client process should be updated to respond to the cancel order from the client and announce this as a failure in the transaction to force this transaction to terminate. The code of the client process should be updated as follows:

```
clientP= BankPortal ⇐ { YIELDD; (login!ID!pass ÷ SKIP) ; YIELDD;
           ( ( (notValid.ErrorMSG ÷ SKIP) ; SKIPP ) □
                   ( (Valid ÷ SKIP) ;  CreditRequest ⇐+ {
            μ.CRrec ( YIELDD ; (request!amount!SEC ÷ SKIP) ; YILEDD ;
                                ( (validBalance ÷ SKIP) ; YIELDD ;
                     ( userOffer □ ( (requestDenied ÷ SKIP) ; SKIPP) ) )
                     □ ( (notvalidBalance ÷ SKIP) ; CRrec) ) } ) }

userOffer= (offer?amount ÷ SKIP) ; YIELDD ; ( ( (AcceptOffer ÷ SKIP) ;
                     YIELDD ; (transferDate?date ÷ SKIP) ; SKIPP)
                ⊓ ( (RejectOffer  ÷ SKIP) ; SKIPP) )

client = clientP ||| (cancel → THROWW)
```

$THROWW$ process is used in this code to announce the failure. Recall that $THROWW$ is the compensable version of the $THROW$ primitive process.

Finally, the system process should be updated to enclose services and processes in a transaction as following:

```
system = client ||A (
   (*BankPortal ||{| ValidID, notValid↑ , Valid↑ |} *Authentication) |||
```

$$\Big( \texttt{*CreditRequest} \parallel_{\texttt{B}} \big( (\texttt{*BalanceValidation} \parallel_{\{|\texttt{analysis,analysed}|\}} \texttt{*SecAnalysis}) \texttt{|||} \texttt{*Rating} \texttt{|||} \texttt{*TransferMoney} \texttt{|||} \texttt{*ClerkList} \texttt{|||} \texttt{*ManagerList} \big) \Big)$$

Where

A={| login, Valid, notValid, request, validBalance, requestDenied,
notvalidBalance, offer, AcceptOffer, RejectOffer, transferDate |}

B={| ValidateBC, validBalance↑ , notvalidBalance↑ , getRate,returnRate,
transMoney ,addtoClist,assessRisk, addtoMlist,Mdecision |}

cFinance= ([system] ||| (Clerks ||| Managers)) $\parallel_{\{|\ a\leftarrow, a\rightarrow\ |\ a\in\ \Sigma\ |\}}$ $\text{B}_{\Sigma}$

The previous adjustments show the importance of extending SOC calculi with compensations. However, to highlight the usability of our improvement to cCSP, we add to the description of the case study the following scenario:

If the request is cancelled after it has been evaluated by a manager then the client should pay an administrative fee because his request has almost finished and has been processed by bank employees.

To implement this scenario we replace the compensation of request atomic process with the variable $X$, and initialise it with $SKIP$, which means initially the client should pay nothing if he cancels his request. However, this compensation will be changed to the atomic process (**pay.(x).admFee**) which should be processed later by the Collecting Department. The code of this scenario should be as following:

X= SKIP

clientP= BankPortal ⇐ { YIELDD; (login!ID!pass ÷ SKIP) ; YIELDD;
( ( (notValid.ErrorMSG ÷ SKIP) ; SKIPP ) □
( (Valid ÷ SKIP) ; CreditRequest $\Leftarrow^{+}$ {
$\mu$.CRrec ( YIELDD ; (request!amount!SEC ÷ X) ; YILEDD ;
( (validBalance ÷ SKIP) ; YIELDD ;
( userOffer □ ( (requestDenied ÷ SKIP) ; SKIPP) ) )
□ ( (notvalidBalance ÷ SKIP) ; CRrec) ) } ) }

Manager= Mlist?>(x).amount.SEC.R.eval ; evaluateM ; X:=pay.(x).admFee ;
x!dec ; Manager

## 8.6 Conclusions and Related Work

General dynamic recovery allows compensations to be replaced or discarded in the compensation sequence. This is useful in cases where compensations are not known from the beginning or if they are subject to change while the system is running.

DEcCSP is a compensating calculus developed as an extension of EcCSP, improving the recovery mechanism from backward recovery to general dynamic recovery, and including all of the CSP standard operators. Due to the time constraints we could not extend the functions of soaCSP in $\text{CSP}_M$ to work with compensable processes and we propose this as future work. The implementation of EcCSP [145] in PAT model checker [131] could be useful.

Compensations are important in modelling complex systems and in the literature compensations have been discussed in different contexts. Considering the aim of this thesis is to develop a SOC calculus, we focus in this section on compensations within the context of SOC calculi.

Compensations are very popular in modelling SOC systems, due to the long running nature of these systems. Static handlers for compensations are considered expressive for most SOC scenarios. For instance, the cancel option of the finance case study which was implemented in this thesis can be implemented without compensation primitives as explained in Section 7.3.2. However, providing such primitives improves the code as shown in Section 8.5, which in turn simplifies the reasoning on such scenarios.

Most SOC calculi introduce static handlers for compensations like COWS calculus [100]. COWS calculus constructs and operators were inspired mainly from BPEL (for BPEL details see Chapter 2), so the static fault and compensation handlers of BPEL scopes have been inherited in COWS.

On the other hand, a number of SOC calculi like the Conversation Calculus [137] and SOCK calculus [80] who have been extended in [51] and [79] respectively to support the dynamic construction of compensation sequences. The developers of the CC calculus [137] encoded cCSP in CC [51] mainly to introduce dynamic compensations. However, the compensating version of CC is not compositional (refer to [96] for more information).

Compensation mechanisms in the compensation extension of SOCK support general dynamic recovery [79]. However, SOCK strictly follows the internet standards' notations and has three designing layers: service behaviours, service engines, and service systems, which in turn affect its simplicity [93].

Additionally, compensations have been introduced in a number of calculi which are

proposed to model web services but do not support session-based communications. For instance, $\text{Web}_\pi$ [99] which supports parallel recovery for statically installed compensations, and $\text{Web}_{\pi\infty}$ [105] which was developed as an improved version of $\text{Web}_\pi$ without time.

General dynamic compensations were introduced to the $\pi$-calculus in [95, 96], however, neither versions supports session-based communications.

Dynamic construction of compensation sequences also has been discussed in the context of more general concurrent calculus. Among the first dynamic compensating calculi for concurrent applications is Structured Activity Compensation (StAC) [48], but StAC needs explicit activation to start the compensations and it introduces many primitives which complicate the usability of the calculus. StAC compensation mechanisms were refined by the works in [49] and [72], but neither of them supports general dynamic recovery.

# 9

# Conclusions and Future work

In this chapter we conclude the thesis and discuss its main results. In addition, we identify the possible directions in which the thesis content could be further extended.

## 9.1 Conclusions and Evaluation

Software architecture and designing methods have evolved from monolithic paradigms to the Service-Oriented Computing (SOC) paradigm, which has been applied successfully to facilitate the integration of systems in distributed environments.

Software applications in the SOC paradigm are called service compositions, and they are defined as an aggregate of loosely coupled autonomous heterogeneous services, which are collectively composed to implement a particular task.

This thesis concerned formal methods support for SOC specifications. More specifically, this thesis applies the theory of process calculi in order to formalise the process of constructing and orchestrating service compositions. The benefit of the formalisation is in producing provably correct models where significant system properties can be checked. One intended application of this approach is the laying of a formal foundation for making SOC systems [67] more trustworthy.

Orchestrating service compositions means to define the operational characteristics of a model; more specifically, it is to define the work flow in such a service composition. Orchestration could be considered similar to designing an activity digram in the UML language. From such orchestration work flow we should be able to identify the routes of messages, services in the composition, and the order of execution of services.

Formal methods support for SOC models by the means of process calculi has been

intensively studied in the literature and we specially single out the SCC calculus [35] which laid the foundation of sessions in the context of SOC calculi. As surveyed in Section 2.1 page 14, several process calculi have been proposed with the aim of improving the notion of sessions.

The SOC calculus proposed in the thesis adapts the notion of sessions and extends it further to address five areas of improvements as stated in the Introduction chapter. These improvements concern the expressivity of the sessions notion, the flexibility in establishing communications, and the computer aid for defining an environment where SOC models can be specified and analysed.

To achieve the final outcome of the thesis, which is soaCSP SOC calculus, the suggested extensions were translated into different issues to be addressed. We preferred to approach the problem of the thesis gradually, so we can study and evaluate each issue separately.

Firstly in Chapter 3 we developed CSPa calculus that supports mixed synchronous and asynchronous communications. The calculus can be described as a buffered version of the standard CSP [124]. The novelty in CSPa is the introduction of *implicit* buffers, which are used in the channel semantics of CSP to facilitate asynchronous communications in a transparent way. We extended CSP syntax to include asynchronous communication primitives, and support these primitives with an operational semantics which explains their behaviour.

To evaluate the content of this chapter we first proved that our buffers do not introduce new traces by proving that the buffered system could be simulated by the system. Additionally, we wanted to study the effects of buffers on CSP. Therefore, we studied the relationship between CSPa and CSP. We conclude that CSPa can be encoded in CSP and that CSPa does not enhance the expressiveness of the calculus. However, we argue that our buffered version of CSP simplifies the reasoning on asynchronous communications. Finally, we studied the effects of these buffers on the termination of CSPa processes, and we proved that buffers in the calculus will not introduce non-termination.

In Chapter 4, we developed $\mathcal{M}$CSP calculus which extends the channel-semantics of CSP to allow channels to carry channel names, and includes mechanisms to change the interface set of parallel compositions as the participants communicate. We extended CSP with mobile channels and we extended CSP operational semantics with new rules to explain the process of updating the interface set of parallel compositions.

Knowing that the $\pi$-calculus is the reference model in mobility, we evaluated our mobility model by encoding the $\pi$-calculus into $\mathcal{M}$CSP. We concluded that the two theories are operationally correspondent.

Following that, in Chapter 5, we developed CSPs calculus, where multiparty sessions can be created, manipulated and terminated transparently without involving the designer. Our calculus comes as a session-based extension of of the standard CSP [124]. CSPs permits multiparty sessions without further grouping notions such as sites or endpoints like in [87, 137]. Sessions are started by invocations, where new invocations could either start a new sub-session or add the invoked service to the current session to establish multiparty session. Communications in CSPs can be interleaved, synchronous, multicast, or broadcast.

We formally introduced this extension by extending the syntax and the operational semantics of CSP. We evaluated our labelled version of CSP by proving that the labels in the calculus will not introduce non-termination, i.e. labels will not prohibit CSPs services from terminating. In addition, we studied the effect of labels on the behaviour of the system. We conclude that labels will not change the behaviour of the system by proving that the behaviour of a labelled service is equivalent to the behaviour of the same service if it is evaluated without labels, then labels are introduced to the behavioural trace afterwards.

In Chapter 6, we developed the final theoretical product of this thesis, soaCSP calculus, which is a compound calculus incorporating the features of CSPa, $\mathcal{M}$CSP, and CSPs calculi. In soaCSP, communications in a multiparty session can be carried out asynchronously or synchronously. In addition, channel names can be sent between services and processes. If a mobile communication is initiated in a session then it carries the state of the session along with the channel name. We extended the syntax of CSP with asynchronous mobile communications, sessioned mobile communications and asynchronous sessioned communications, and we updated the operational semantics of CSP accordingly.

The features of CSPa, $\mathcal{M}$CSP, and CSPs have been evaluated separately. Therefore, in this chapter we wanted to test whether our compound calculus, soaCSP, is expressive enough for BPEL [14] by encoding the syntax of BPEL into the syntax of soaCSP. We conclude that, if we exclude timed constructs from BPEL, as soaCSP does not admit timed events, then soaCSP is expressive enough for BPEL.

Furthermore, in Chapter 7 we implemented soaCSP in $\text{CSP}_M$, the machine readable version of CSP, which is used as the input language for a range of tools. To the best of our knowledge, in the context of SOC calculi, the implementation of soaCSP in FDR is the first to provide a computer-based environment to reason on mobile processes and multiparty sessions.

In addition, we demonstrated an implementation of the finance case study [65, 66, 132] from the Sensoria project [8] in our calculus. We remark that the produced code is concise compared to the description of the finance case study in BPEL [66] or in COWS [132].

This is due to the fact that, soaCSP provides an expressive communication model, which resulted in fewer services and communications for circulating data.

We also remark that soaCSP can model processes as well as services. This was helpful when modelling objects in the case study which were not necessarily a service such as bank employees. Therefore, in soaCSP we model bank employees as ordinary processes and we permit these processes to communicate with the running session when needed. This feature produces an interesting result from the soaCSP models, this being able to model and reason on the communications between legacy systems and SOC systems without transform the legacy system into services.

Furthermore in the chapter, we ran the implementation in Probe (the trace animator of CSP) mainly to show that the produced traces show the desired output. Following that, we ran the implementation in FDR to demonstrate the reasoning mechanisms of soaCSP (inherited from CSP), and most importantly, to prove that our implementation will not deadlock or diverge.

Finally, in Chapter 8 we proposed an improvement to the current version of compensating CSP in order to facilitate general dynamic recovery, and we proposed incorporating this in soaCSP in future work, as this will support the calculus with an expressive exception system as demonstrated in Section 8.5.

By far, we have demonstrated the expressivity of the soaCSP model, and emphasised the design capabilities of the calculus. We also have shown that the implementation of soaCSP in $\text{CSP}_M$ provides a computer-based environment in which to model SOC systems and reason on these systems to check a range of desirable properties including deadlock-freedom and divergence-freedom. In the related work sections of each chapter we highlighted the novelty of our solution compared to other formal models.

We conclude with a discussion of the feasibility of soaCSP to take the place of BPEL in the industry. The thesis shows that the design capabilities of our calculus nominate soaCSP to strongly compete with BPEL as a modelling language for SOC systems. Especially because soaCSP provides a computer-based environment for reasoning on SOC models, a feature not supported in BPEL.

However, apart from having support from major companies in the industry like IBM, Microsoft, and Oracle, the power of BPEL is in the realization step. The BPEL modelling language is an XML-based language, which means that it is a human readable and machine readable language. Therefore, it can be used to write computer readable specifications, which can be put up for running in a very short time. This is achievable by defining a number of implementation attributes which identify such items as the URL of services.

Then, input these specifications to one of the BPEL engines which will turn them to an executable code on the internet. In addition, BPEL is supported with graphical modelling languages which enhance the usability of the calculus.

Therefore, to enhance the chances of soaCSP to compete in the market with BPEL, in the next section we propose to support soaCSP with a graphical modelling language, and to encode a model transformation from BPEL to soaCSP and vice versa to facilitate the realization of service composition.

## 9.2 Future Work

The content of this thesis leaves many avenues which could be proposed for future work. We classify these avenues into three categories: work in progress, future extensions, and desirable enhancements. We list the work in these categories on order of importance.

**Work in progress** This category contains the following ongoing works.

- **soaCSP denotational semantics:** In Chapter 7 we provided an implementation of soaCSP in FDR. This implementation could be used to reason on the correctness of soaCSP scripts using the denotational models of the standard CSP. The denotational models of CSP provide simple proof techniques to assert the conformance between specification and implementation, deadlock-freedom, and divergence-freedom, in addition to determinism and bisimulations.

  Defining a denotational semantics for soaCSP will permit designers to reason theoretically on these important properties. Additionally, having another semantics will prove the correctness of our operational semantics, if we could prove the correspondence between the two models. Therefore, in [21] we developed a trace semantics for CSPs and we proved the correspondence between the two semantics. This provides a theoretical model to reason on good/bad traces as in the original CSP [84].

  We aim to develop a trace semantics for the remaining of soaCSP calculus, and to develop the other behavioural models of CSP: stable failure model and divergence failure model for soaCSP. These models will enrich the reasoning power of the calculus by enabling designers to verify additional behavioural properties like deadlock-freedom and divergence-freedom.

- **Compensations:** In Chapter 8 we extended cCSP with primitives to model general dynamic recovery, and we demonstrated theoretically the result of incorporating the

failure termination algorithm of DEcCSP in soaCSP. This extension will support soaCSP with an expressive exception system. A preliminary model is available[1].

- **soaCSP implementation:** In Chapter 7 we implemented soaCSP into $CSP_M$, the input language of FDR. However, as clarified in Section 7.3.3 this implementation does not include the feature of carrying session labels with channel names, when communicating mobile channels, as we suggested in the theoretical model of soaCSP; we are working to implement this feature. In addition, we are working to force sessions to close in order instead of the parallel closures in the current implementation.

  FDR is not an open source model checker, and its input language is a machine readable version of CSP. Therefore, our implementation could be considered as informal encoding of our calculus into CSP. However, this compromises the usability of our calculus by replacing operators with a sequence of function calls. Alternatively, implementing soaCSP in another model checker which supports CSP models and in the same time provides the source code or coding in the source language will be investigated; PAT model checker [131] was suggested by Zhenbang Chen, the author of the Extended cCSP model checker [145].

**Future extensions** In this category we highlight the extensions below, which we consider important for enhancing the usability and the functionality of the calculus:

- **BPEL model transformation:** In Chapter 6 we discussed the relationship between soaCSP and BPEL. A detailed encoding supported by a model transformation from BPEL into soaCSP will be an interesting extension. Although this model will not allow the users to access the new features in soaCSP, it will encourage BPEL users to use soaCSP as a validation platform for their specifications.

- **Model transformation to UML4SOA:** Supporting the calculus with a graphical modelling language like UML4SOA [11] or new defined notations, in order to provide a graphical environment for specifying SOC models in soaCSP calculus, is a desirable extension. Model transformation from UML4SOA language to our calculus and via versa should be implemented in parallel; previous work conducted by Yeung in [143] which transforms UML diagrams to CSP scripts could be useful.

- **Correct models by construction:** Models in soaCSP are provably correct, i.e. the reasoning mechanisms of the calculus should be used to prove that models are seman-

[1]Visit: http://fac.ksu.edu.sa/sites/default/files/fullSoaCSP4v.pdf

tically correct. An alternative approach is to use types to govern the construction of models to ensure that models are correct by construction. For instance, type systems are suggested in [87] and also in subsequent works by the authors to design correct SOC models by construction. Type systems provide theoretical models to discuss features like protocol fidelity and static analysis of models. Moreover, theorem provers supporting types like Coq [31], could be used to provide a computer-based environment for constructing correct models. For instance, the Coq theorem prover was used in [118] to construct correct model transformations. As a start, we implemented CSP operational semantics in Coq[2], which shows the feasibility of this future work.

Implementing soaCSP in Coq will help in extracting code from soaCSP models. This because, Coq provides the feature of extracting codes to Haskell or to Ocaml functional languages.

**Desirable enhancements** In this category we propose the extensions below, which we found interesting and could be considered as future work.

- **Choreographies:** In this thesis we extended CSP with orchestration primitives. As a further extension, soaCSP could be extended to define a global scenario, as in WS-CDL [13], then for implementation purposes this scenario could be projected into soaCSP services as in [87]. Especially because soaCSP can define ordinary processes. As a result, the equational reasoning and refinement theory of CSP could be used to reason on choreography scenarios.

- **Service discovery and services contracts**: This thesis concerns the orchestration of services assuming that services are selected. However, studying the process of publishing services' descriptions (contracts), then discovering services according to these contracts, as proposed in [55, 41], is an interesting extension.

- **Java interpreter for soaCSP:** JCSP [138] is a java interpreter for CSP models. Extending JCSP to include the new primitives of soaCSP is an interesting extension. Especially because in the industry, JAVA is the underlying programming language for most services.

- **Enhance the operational semantics of soaCSP**: In this thesis we pointed out two features to enhance the soaCSP operational semantics. The first feature is extending

[2]Visit: http://fac.ksu.edu.sa/sites/default/files/soacspincoq.pdf

the operational semantics to show the order and effects of multi-components events. The second feature is extending the operational semantics to allow all combinations of mobile channels to synchronise.

- **Extending soaCSP with time**: The encoding from BPEL into soaCPS in Section 6.3 shows that although soaCSP has more expressive features in terms of creating sessions and establishing communications, soaCSP is not enough for BPEL because soaCSP does not support time. Therefore, a timed version of soaCSP would be useful. The Timed CSP [104] could be used as a guidance for introducing time into soaCSP.

- **Extending soaCSP with QoS attributes**: Verifying non-functional requirements is critical in applications where post-testing is hard like SOC models. For this reason some SOC calculi have been extended with stochastics and probabilities to enable non-functional requirements to be quantified in order to reason on them. For instance, COWS web calculus is further extended into stochastic COWS (sCOWS) [119] in which actions are associated with variables express their weights for quantitative analysis. In addition, CaSPiS [36] is further extended into stochastic CaSPiS (MarCaSPiS) [62], which extended CaSPiS with Markovian rules [32] to facilitate quantitative analysis for QoS requirements in services.

  Improving the functionality of soaCSP by introducing primitives to quantify Quality of Service attributes in order to formally specify and reason on the performance characteristic of the calculus would be a useful extension.

# Bibliography


[1] Apache ODE. http://ode.apache.org/. Online; accessed 7 September 2015.

[2] CORBA 3.3. http://www.omg.org/spec/CORBA/3.3/. Online; accessed 7 September 2015.

[3] Distributed Component Object Model (DCOM) Remote Protocol Specification. https://msdn.microsoft.com/library/cc201989.aspx. Online; accessed 7 September 2015.

[4] Java EE enterprise standard. https://www.oracle.com/java/technologies/java-ee.html. Online; accessed 7 September 2015.

[5] Jolie website. http://www.jolie-lang.org/. Online; accessed 7 September 2015.

[6] Oracle BPEL Process Manager. http://www.oracle.com/technetwork/middleware/bpel/overview/index.html. Online; accessed 7 September 2015.

[7] ProBE Manual. http://www.fsel.com/documentation/probe/probe-doc-html/html/index.html. Online; accessed 7 September 2015.

[8] Sensoria. http://www.sensoria-ist.eu. Online; accessed 7 September 2015.

[9] Simple Object Access Protocol (SOAP) Version 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. Online; accessed 7 September 2015.

[10] The ActiveBPEL Server. http://www.activevos.com/content/developers/education/sample_active_bpel_admin_api/doc/index.html. Online; accessed 7 September 2015.

[11] UML4SOA. http://www.uml4soa.eu. Online; accessed 7 September 2015.

[12] FDR2 Manual. http://www.fsel.com/software.html, 1992-2009. Online; accessed 7 September 2015.

[13] Web Services Choreography Description Language Version 1.0 W3C Candidate Recommendation 9 November 2005. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, 2005. Online; accessed 7 September 2015.

[14] Web Services Business Process Execution Language Version 2.0 OASIS Standard 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007. Online; accessed 7 September 2015.

[15] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language W3C Recommendation 26 June 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626/, 2007. Online; accessed 7 September 2015.

[16] Web Services Coordination (WS-Coordination) Version 1.2. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html, February 2009. Online; accessed 7 September 2015.

[17] A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] A. S. Al-Humaimeedy and M. Fernández. General Dynamic Recovery for Compensating CSP. In B. Löwe and G. Winskel, editors, *DCM*, volume 143 of *EPTCS*, pages 3–16, 2014.

[19] A. S. Al-Humaimeedy and M. Fernández. Introducing Mobility into CSP. In *NWPT 2014, Halmstad University, Sweden, extended abstract (full version submitted to JLAMP)*, 2014.

[20] A. S. Al-Humaimeedy and M. Fernández. Enabling Synchronous and Asynchronous Communications in CSP for SOC. *Electr. Notes Theor. Comput. Sci.*, 312:69–88, 2015.

[21] A. S. Al-Humaimeedy and M. Fernández. Enhancing the Specification and Verification Techniques of Multiparty Sessions in SOC. In M. Falaschi and E. Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 19–30. ACM, 2015.

[22] K. Bae and J. Meseguer. A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. *Electr. Notes Theor. Comput. Sci.*, 290:19–36, 2012.

[23] J. C. M. Baeten. A Brief History of Process Algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[24] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[25] C. Baier and J. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[26] M. Bartoletti, L. Caires, I. Lanese, F. Mazzanti, D. Sangiorgi, H. Vieira, and R. Zunino. Tools and Verification. In M. Wirsing and M. Hlzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, pages 408–427. Springer Berlin Heidelberg, 2011.

[27] R. Beauxis, C. Palamidessi, and F. D. Valencia. On the Asynchronous Nature of the Asynchronous pi-Calculus. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 473–492. Springer, 2008.

[28] A. Bejleri and N. Yoshida. Synchronous Multiparty Session Types. *Electr. Notes Theor. Comput. Sci.*, 241:3–33, 2009.

[29] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process Algebra with Asynchronous Communication Mechanisms. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 1984.

[30] M. Bernardo, L. Padovani, and G. Zavattaro, editors. *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, volume 5569 of *Lecture Notes in Computer Science*. Springer, 2009.

[31] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, Berlin, New York, 2004.

[32] F. Bloch and D. Cantala. Markovian Assignment Rules. Working Papers hal-00356304, HAL, Nov 2008.

[33] L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[34] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 240–256. Springer Berlin Heidelberg, 2008.

[35] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: A Service Centered Calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[36] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2008.

[37] L. Bourgé. On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes. *Acta Inf.*, 25(2):179–201, Feb. 1988.

[38] M. Bravetti, M. Carbone, T. T. Hildebrandt, I. Lanese, J. Mauro, J. A. Pérez, and G. Zavattaro. Towards Global and Local Types for Adaptation. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, volume 8368 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2013.

[39] M. Bravetti, L. Kloul, and G. Zavattaro, editors. *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, volume 3670 of *Lecture Notes in Computer Science*. Springer, 2005.

[40] M. Bravetti, M. Núñez, and G. Zavattaro, editors. *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*. Springer, 2006.

[41] M. Bravetti and G. Zavattaro. A Theory for Strong Service Compliance. In A. L. Murphy and J. Vitek, editors, *COORDINATION*, volume 4467 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2007.

[42] M. Bravetti and G. Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In M. Lumpe and W. Vanderperren, editors, *Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.

[43] R. Bruni, M. J. Butler, C. Ferreira, C. A. R. Hoare, H. C. Melgratti, and U. Montanari. Comparing Two Approaches to Compensable Flow Composition. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.

[44] R. Bruni, A. Kersten, I. Lanese, and G. Spagnolo. A New Strategy for Distributed Compensations with Interruption in Long-Running Transactions. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques - 20th International Workshop, WADT 2010, Etelsen, Germany, July 1-4, 2010, Revised Selected Papers*, volume 7137 of *Lecture Notes in Computer Science*, pages 42–60. Springer, 2010.

[45] R. Bruni, I. Lanese, H. C. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[46] R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In J. Palsberg and M. Abadi, editors, *POPL*, pages 209–220. ACM, 2005.

[47] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration Conformance for System Design. In *In COORDINATION, volume 4038 of LNCS*, pages 63–81. Springer, 2006.

[48] M. J. Butler and C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions. In R. De Nicola, G. L. Ferrari, and G. Meredith, editors, *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.

[49] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In Abdallah et al. [17], pages 133–150.

[50] M. J. Butler and S. Ripon. Executable Semantics for Compensating CSP. In Bravetti et al. [39], pages 243–256.

[51] L. Caires, C. Ferreira, and H. T. Vieira. A Process Calculus Analysis of Compensations. In Kaklamanis and Nielson [88], pages 87–103.

[52] M. Carbone, K. Honda, and N. Yoshida. A Calculus of Global Interaction based on Session Types. *Electr. Notes Theor. Comput. Sci.*, 171(3):127–151, 2007.

[53] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centered Programming for Web Services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

[54] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[55] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A Formal Account of Contracts for Web Services. In Bravetti et al. [40], pages 148–162.

[56] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of Session Types. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 219–230. ACM, 2009.

[57] Z. Chen and Z. Liu. An Extended cCSP with Stable Failures Semantics. In A. Cavalcanti, D. Déharbe, M. Gaudel, and J. Woodcock, editors, *Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings*, volume 6255 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2010.

[58] Z. Chen, Z. Liu, and J. Wang. Failure-Divergence Semantics and Refinement of Long Running Transactions. *Theor. Comput. Sci.*, 455:31–65, 2012.

[59] V. Danos and J. Krivine. Reversible Communicating Systems. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[60] F. S. de Boer, J. W. Klop, and C. Palamidessi. Asynchronous Communication in Process Algebra. In *LICS*, pages 137–147, 1992.

[61] R. De Nicola, editor. *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April*

*1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.

[62] R. De Nicola, D. Latella, M. Loreti, and M. Massink. MarCaSPiS: a Markovian Extension of a Calculus for Services. *Electr. Notes Theor. Comput. Sci.*, 229(4):11–26, 2009.

[63] E. de Vries, V. Koutavas, and M. Hennessy. Communicating Transactions - (Extended Abstract). In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 569–583. Springer, 2010.

[64] G. Díaz, J. J. Pardo, M. Cambronero, V. Valero, and F. Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata. In Bravetti et al. [39], pages 230–242.

[65] J. Elgner, S. Gnesi, N. Koch, and P. Mayer. Introduction to the Sensoria Case Studies. In M. Wirsing and M. M. Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lecture Notes in Computer Science*, pages 26–34. Springer, 2011.

[66] J. Elgner and K. Swierkot. Finance Case Study Credit Portal. Technical Report 016004, Sensoria, January 31, 2010.

[67] T. Erl. *Service-Oriented Architecture: Concepts, Technology and Design*. Printice-Hall, 2005.

[68] M. Fernández. *Programming Languages and Operational Semantics*. King's College London, 2004.

[69] B. A. Forouzan. *TCP/IP Protocol Suite*. McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2002.

[70] C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.

[71] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514. Springer, 2004.

[72] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 249–259. ACM Press, 1987.

[73] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-Consistent Reversible Debugging. In S. Gnesi and A. Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[74] T. Gibson-Robinson and M. Goldsmith. The Meaning and Implementation of SKIP in CSP. In *Communicating Process Architectures 2013*, 2013.

[75] M. Giunti, K. Honda, V. T. Vasconcelos, and N. Yoshida. Session-Based Type Discipline for $\pi$-calculus with Matching. *In the preproceedings of PLACES*, 9, 2009.

[76] A. D. Gordon, editor. *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*. Springer, 2010.

[77] D. Gorla. Towards a Unified Approach to Encodability and Separation Results for Process Calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[78] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[79] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the Interplay Between Fault Handling and Request-Response Service Invocations. In J. Billington, Z. Duan, and M. Koutny, editors, *8th International Conference on Application of Concurrency to System Design (ACSD 2008), Xi'an, China, June 23-27, 2008*, pages 190–198. IEEE, 2008.

[80] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In A. Dan and W. Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[81] J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`.

[82] B. Harvey. *Computer Science Logo Style Volume 3: Beyond Programming.* MIT Press, 1997.

[83] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 220–235, 2005.

[84] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[85] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[86] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In P. America, editor, *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[87] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[88] C. Kaklamanis and F. Nielson, editors. *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*. Springer, 2009.

[89] R. Kazhamiakin, P. K. Pandya, and M. Pistore. Timed Modelling and Analysis in Web Service Compositions. In *Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, The International Dependability Conference - Bridging Theory and Practice, April 20-22 2006, Vienna University of Technology, Austria*, pages 840–846. IEEE Computer Society, 2006.

[90] M. Khaxar, S. Jalili, N. Khakpour, and M. S. Jokhio. Monitoring Safety Properties of Composite Web Services at Runtime Using CSP. In *Workshops Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference, EDOCw 2009, 1-4 September 2009, Auckland, New Zealand*, pages 107–113. IEEE Computer Society, 2009.

[91] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc Programming Language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.

[92] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In A. Cerone and S. Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 323–332. IEEE Computer Society, 2008.

[93] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 305–314. IEEE Computer Society, 2007.

[94] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-Consistent Reversibility. *Bulletin of the EATCS*, 114, 2014.

[95] I. Lanese, C. Vaz, and C. Ferreira. On the Expressive Power of Primitives for Compensation Handling. In Gordon [76], pages 366–386.

[96] I. Lanese, C. Vaz, and C. Ferreira. On the Expressive Power of Primitives for Compensation Handling (Journal version). 2011.

[97] I. Lanese and G. Zavattaro. Programming Sagas in SOCK. In D. V. Hung and P. Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 189–198. IEEE Computer Society, 2009.

[98] C. Laneve and L. Padovani. Smooth Orchestrators. In L. Aceto and A. Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2006.

[99] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In V. Sassone, editor, *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[100] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In De Nicola [61], pages 33–47.

[101] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.

[102] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin Heidelberg, 2009.

[103] H. A. López and J. A. Pérez. Time and Exceptional Behavior in Multiparty Structured Interactions. In M. Carbone and J. Petit, editors, *Web Services and Formal Methods - 8th International Workshop, WS-FM 2011, Clermont-Ferrand, France, September 1-2, 2011, Revised Selected Papers*, volume 7176 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2011.

[104] G. Lowe. *Probabilities and Priorities in Timed CSP*. PhD thesis, Oxford University, 1993.

[105] M. Mazzara and I. Lanese. Towards a Unifying Theory for Web Services Composition. In Bravetti et al. [40], pages 257–272.

[106] L. G. Mezzina. *Typing Services*. PhD thesis, IMT Institute for Advanced Studies, Lucca, 2009.

[107] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[108] R. Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[109] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I & II. *Inf. Comput.*, 100(1):1–77, 1992.

[110] F. Montesi and N. Yoshida. Compositional Choreographies. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013.

[111] U. Nestmann and B. C. Pierce. Decoding Choice Encodings. *Inf. Comput.*, 163(1):1–59, 2000.

[112] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[113] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous $\pi$-calculus. *CoRR*, cs.PL/9809008, 1998.

[114] R. M. Pessoa, E. G. da Silva, M. van Sinderen, D. A. C. Quartel, and L. F. Pires. Enterprise Interoperability with SOA: a Survey of Service Composition Approaches. In M. van Sinderen, J. P. A. Almeida, L. F. Pires, and M. Steen, editors, *Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, ECOCW 2008, 16 September 2008, Munich, Germany*, pages 238–251. IEEE Computer Society, 2008.

[115] K. Peters and R. J. van Glabbeek. Analysing and Comparing Encodability Criteria. In S. Crafa and D. Gebler, editors, *Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015.*, volume 190 of *EPTCS*, pages 46–60, 2015.

[116] I. C. C. Phillips and I. Ulidowski. Reversing Algebraic Process Calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007.

[117] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[118] I. Poernomo and J. Terrell. Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2010.

[119] D. Prandi and P. Quaglia. Stochastic COWS. In B. J. Krämer, K. Lin, and P. Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.

[120] S. Ripon. *Extending and Relating Semantic Models of Compensating CSP.* PhD thesis, University of Southampton, 2008.

[121] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall (Pearson), 1997 revised 2005.

[122] A. W. Roscoe. On the Expressiveness of CSP, 2008. Available from: `https://www.cs.ox.ac.uk/files/1383/complete(3).pdf`.

[123] A. W. Roscoe. CSP is expressive enough for $\pi$. *Reflections on the work of C.A.R. Hoare, Springer 2010*, 2010.

[124] A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.

[125] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*, page 43. IEEE Computer Society, 2004.

[126] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis.

[127] D. Sangiorgi and D. Walker. *The Pi-Calculus a Theory of Mobile Processes.* Cambridge University Press, 2001.

[128] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, Oxford University, 1998.

[129] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[130] I. Sommerville. *Software Engineering*. Pearson Education Limited, 2007.

[131] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[132] F. Tiezzi. *Specification and Analysis of Service-Oriented Applications*. PhD thesis, Dipartimento di Sistemi e Informatica, Universita degli Studi di Firenze, April 2009.

[133] F. Tiezzi and N. Yoshida. Towards Reversible Sessions. In A. F. Donaldson and V. T. Vasconcelos, editors, *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014.*, volume 155 of *EPTCS*, pages 17–24, 2014.

[134] B. Vajar, S. Schneider, and H. Treharne. Mobile CSP||B. *ECEASST*, 23, 2009.

[135] C. Vaz, C. Ferreira, and A. Ravara. Dynamic Recovering of Long Running Transactions. In Kaklamanis and Nielson [88], pages 201–215.

[136] H. T. Vieira. *A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing*. PhD thesis, The New University of Lisbon, 2010.

[137] H. T. Vieira, L. Caires, and J. C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In S. Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[138] P. Welch and N. Brown. The JCSP Home Page. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`, 1999. Online; accessed 7 September 2015.

[139] P. H. Welch and F. R. M. Barnes. Communicating Mobile Processes. In Abdallah et al. [17], pages 175–210.

[140] P. H. Welch and F. R. M. Barnes. A CSP Model for Mobile Channels. In P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *The thirty-first Communicating Process Architectures*

*Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 17–33. IOS Press, 2008.

[141] F. Wiedijk. Comparing Mathematical Provers. In A. Asperti, B. Buchberger, and J. H. Davenport, editors, *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003.

[142] M. Wirsing, G. Carizzon, S. Gilmore, L. Gnczy, N. Koch, P. Mayer, and C. Palasciano. A Systematic Approach to Developing Service-Oriented Systems. *Sensoria White Paper*, 2007.

[143] W. L. Yeung. A Dual-Formalism Approach to Checking Consistency of Class and State Diagrams in UML. In J. C. Augusto and U. Ultes-Nitsche, editors, *Verification and Validation of Enterprise Information Systems, Proceedings of the 2nd International Workshop on Verification and Validation of Enterprise Information Systems, VVEIS 2004, In conjunction with ICEIS 2004, Porto, Portugal, April 2004*, pages 2–9. INSTICC Press, 2004.

[144] W. L. Yeung. CSP-Based Verification for Web Service Orchestration and Choreography. *Simulation*, 83(1):65–74, 2007.

[145] H. Yu, Z. Chen, and J. Wang. An Operational Semantics for Model Checking Long Running Transactions. In E. Tuosto and C. Ouyang, editors, *Web Services and Formal Methods - 10th International Workshop, WS-FM 2013, Beijing, China, August 2013, Revised Selected Papers*, volume 8379 of *Lecture Notes in Computer Science*, pages 168–187. Springer, 2013.